Updated 2/15/2023

# Topic 1: Data models and query languages
# Unit 4: Datalog
# Lecture 10

Wolfgang Gatterbauer

CS7240 Principles of scalable data management (sp23)

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

2/10/2023

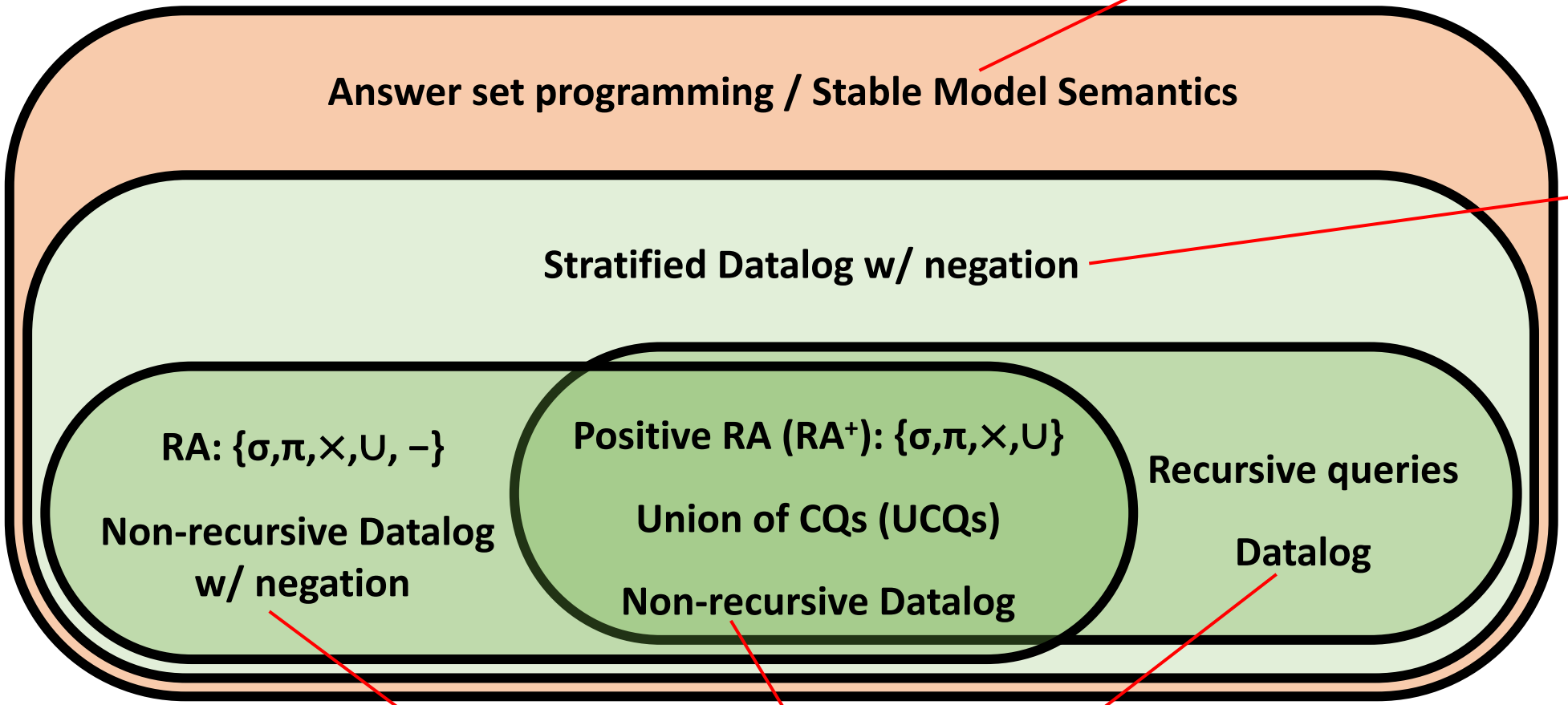
# Pre-class conversations

- Last class summary
- Project ideas
- Explaining the chase procedure

- today:
  - Adding negation to Datalog. What can go wrong ...

# Hierarchy of expressiveness

ASP can express NP-complete problems
(For Turing-completeness, we would only have to add functions, i.e. the ability to create new values not previously found in the EDB)




can express all polynomial time queries on ordered databases relying on only information encoded in tables (e.g. excludes arithmetical functions)

Notice that Datalog and UCQs often assume an unordered domain and no built-in predicates. For equality, we assume here an ordered domain and allow built-in predicates (>,<,≤,≥,!=).

# Negation in Datalog

- Various semantics have been proposed for supporting negation in Datalog that still allow tractability
- We will first look at two:
  - 1. Semipositive Datalog¬ (restricted): PTIME
  - 2. Stratified Datalog¬ (standard): PTIME
- We will later look at a more powerful (but intractable) semantics
  - Stable Models semantics (or answer set programming ASP): NP-complete

# 1. Semipositive Programs and Safety ~ ¬ ! NOT

```
Friend(x,y) :- Likes(x,y), !Parent(y,x).
```

$$\text{Likes} - \pi_{y,x}\text{Parent}$$

A semipositive program is a program where only EDBs may be negated

- Semantics: same as ordinary Datalog programs
- Safety: rule is safe if every variable occurs in a positive (= unnegated) relational atom (ensures domain independence: the results of programs are finite and depend only on the actual contents of the database)

Exercise: Are following rules safe?

```
S(x) :- T(y), Arc(z,y), !Arc(x,y).
```

?

```
S(x) :- T(y), !T(x).
```

?

# 1. Semipositive Programs and Safety

```
Friend(x,y) :- Likes(x,y), !Parent(y,x).
```

$$\text{Likes} - \pi_{y,x}\text{Parent}$$

A semipositive program is a program where only EDBs may be negated

- Semantics: same as ordinary Datalog programs
- Safety: rule is safe if every variable occurs in a positive (= unnegated) relational atom (ensures domain independence: the results of programs are finite and depend only on the actual contents of the database)

Exercise: Are following rules safe?

```
S(x) :- T(y), Arc(z,y), !Arc(x,y).
```

unsafe

```
S(x) :- T(y), !T(x).
```

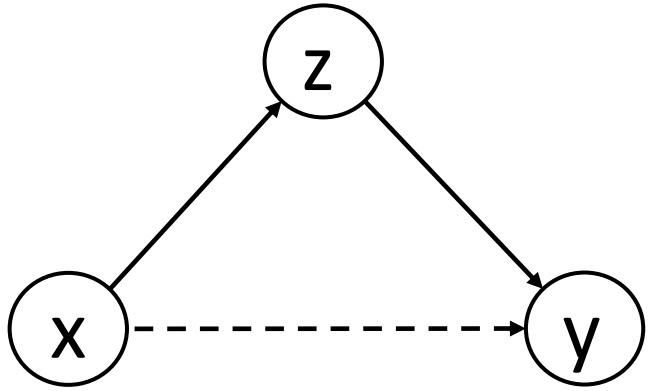
unsafe

# 1. Semipositive: Negated Atoms

Arc(Source,Target)

- We may put !, ¬, ~, or not in front of an atom to negate its meaning.
- EXAMPLE: Return all pairs of nodes (x,y) where y is two hops away from x, but not an immediate neighbor of x.

?

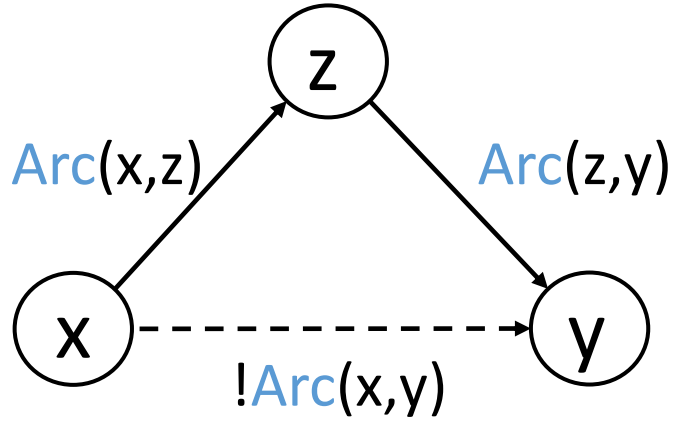


# 1. Semipositive: Negated Atoms

Arc(Source,Target)

- We may put !, ¬, ~, or not in front of an atom to negate its meaning.
- EXAMPLE: Return all pairs of nodes (x,y) where y is two hops away from x, but not an immediate neighbor of x.

```
TwoHopsAway(x,y) :- Arc(x,z), Arc(z,y), !Arc(x,y).
```




A(S,T)

501

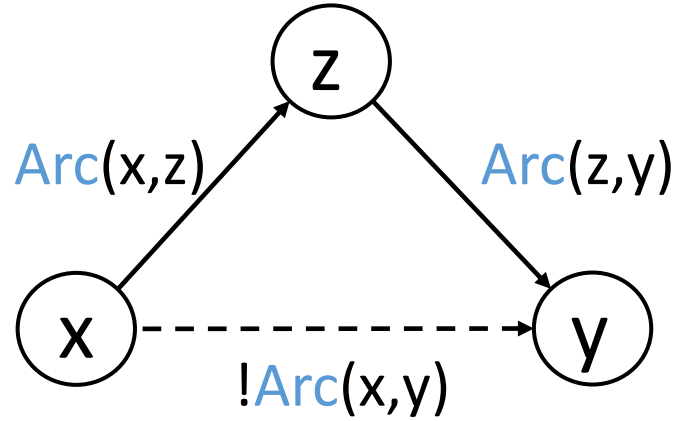
SQL ?

# 1. Semipositive: Negated Atoms

Arc(Source,Target)

- We may put !, ¬, ~, or not in front of an atom to negate its meaning.
- EXAMPLE: Return all pairs of nodes (x,y) where y is two hops away from x, but not an immediate neighbor of x.

```
TwoHopsAway(x,y) :- Arc(x,z), Arc(z,y), !Arc(x,y).
```




A(S,T)

501

```
SELECT A1.S, A2.T
FROM A A1, A A2
WHERE A1.T = A2.S
AND NOT EXISTS
    (SELECT *
    FROM A A3
    WHERE A3.S = A1.S
    AND A3.T = A2.T)
```

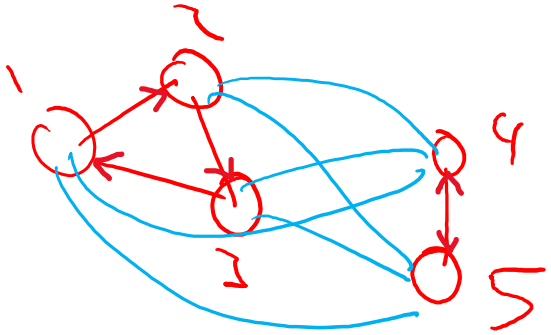
# Exampe: beyond Semipositive

Arc(Source,Target)
Node(id)

Compute all pairs of disconnected nodes in a graph.

?

Node is basically ADom:
Node(x) :- Arc(x,y)
Node(y) :- Arc(x,y)

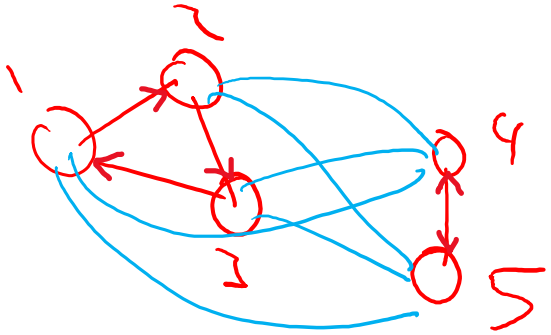


# Exampe: beyond Semipositive

Arc(Source,Target)
Node(id)

Compute all pairs of disconnected nodes in a graph.

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

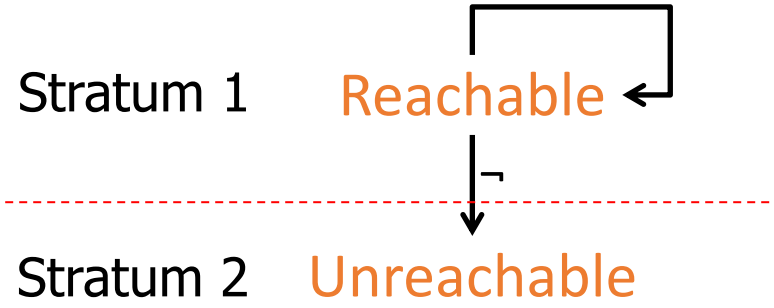


# Exampe: beyond Semipositive

Arc(Source,Target)
Node(id)

Compute all pairs of disconnected nodes in a graph.

```
Reachable(x,y) :- Arc(x,y).
Reachable(x,y) :- Arc(x,z), Reachable(z,y).
Unreachable(x,y) :- Node(x), Node(y), !Reachable(x,y).
```




- Straightforward syntactic restriction.
- When the Datalog program is stratified, we can evaluate IDB predicates stratum-by-stratum
- Once evaluated, treat it as EDB for higher strata.

Precedence graph
- Nodes = IDB predicates
- Arc p→q if predicate q depends on p
- Label this arc "¬" if predicate p is negated

think: "topological sort"

Non-stratified example:

```
LeftBox(x) :- !LeftBox(x), Item(x).
```

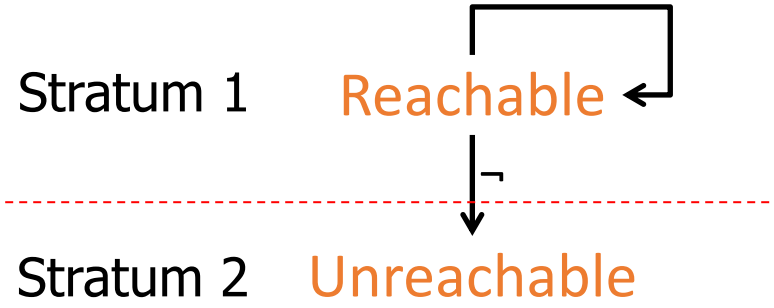
?

# Exampe: beyond Semipositive

Arc(Source,Target)
Node(id)

Compute all pairs of disconnected nodes in a graph.

```
Reachable(x,y) :- Arc(x,y).
Reachable(x,y) :- Arc(x,z), Reachable(z,y).
Unreachable(x,y) :- Node(x), Node(y), !Reachable(x,y).
```



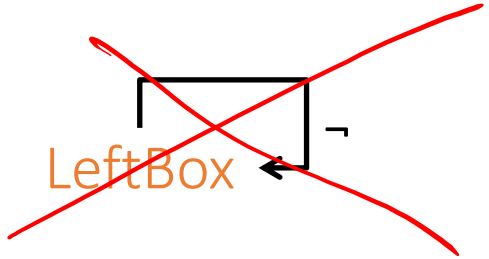

- Straightforward syntactic restriction.
- When the Datalog program is stratified, we can evaluate IDB predicates stratum-by-stratum
- Once evaluated, treat it as EDB for higher strata.

Precedence graph
- Nodes = IDB predicates
- Arc p→q if predicate q depends on p
- Label this arc "¬" if predicate p is negated

think: "topological sort"

Non-stratified example:

```
LeftBox(x) :- !LeftBox(x), Item(x).
```



# 2. Stratified Programs: Definition and Semantics

- DEFINITION: Let **P** be a Datalog program, **E** be the set of EDB predicates, and **I** be the set of IDB predicates. A stratification of **P** is a partitioning of the IDB predicates into disjoint sets $I_1,\ldots,I_k$ such that:
  - For i=1,...,k, every rule with head in $I_i$ has possible body predicates only from **E**, $I_1,\ldots, I_i$
  - For i=1,...,k, every rule with head in $I_i$ has negated body predicates only from **E**, $I_1,\ldots, I_{i-1}$

- SEMANTICS:
  - For i=1,...,k:
    - Compute the IDBs of the stratum $I_i$, possibly via recursion
    - Add computed IDBs to the EDBs
  - Due to the definition of stratification, each $E_i$ can be viewed as semipositive

# 2. Theorems on Stratification

- THEOREM 1: A program has a stratification if and only if its dependency graph does not contain a cycle with a "negated edge"
  - Dependency graph is defined as previously, except that edges can be labeled with negation
  - Hence, we can test for stratifiability efficiently, via graph reachability

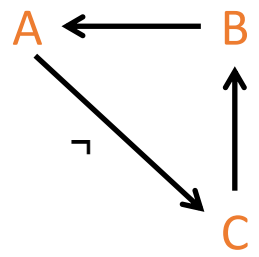
```
A(x) :- B(x).
B(x) :- C(x).
C(x) :- ¬A(x).
```

Can it be stratified ?

# 2. Theorems on Stratification

- THEOREM 1: A program has a stratification if and only if its dependency graph does not contain a cycle with a "negated edge"
  - Dependency graph is defined as previously, except that edges can be labeled with negation
  - Hence, we can test for stratifiability efficiently, via graph reachability

```
A(x) :- B(x).
B(x) :- C(x).
C(x) :- ¬A(x).
```




- THEOREM 2: Non-recursive Datalog with negation can always be stratified via the topological order

- THEOREM 3: Non-recursive Datalog with negation has the same expressive power as the algebra $\{\sigma_{=}, \pi, \times, \cup, -\}$
  - Extendable to RA if we add the comparison predicates <, >, !=, <=, >=

# 2. Stratification practice

Parent(P,C)

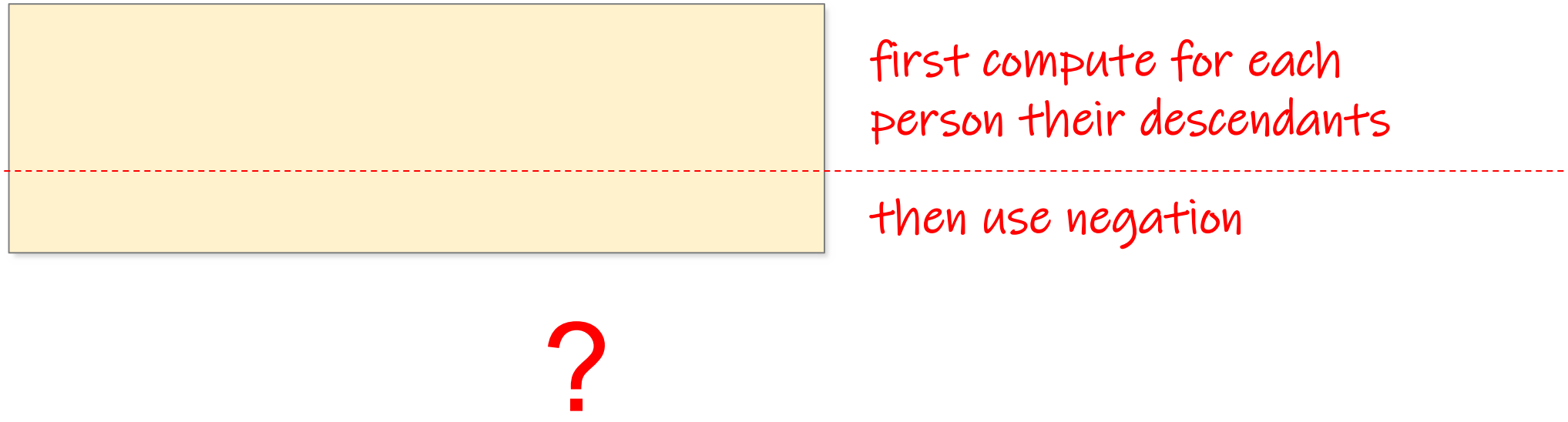
Q: Find all descendants of Alice,
who are not descendants of Bob

?

# 2. Stratification practice

Parent(P,C)

Q: Find all descendants of Alice,
who are not descendants of Bob



first compute for each person their descendants

then use negation

?

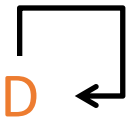
# 2. Stratification practice

Parent(P,C)

Q: Find all descendants of Alice,
who are not descendants of Bob

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

first compute for each person their descendants




then use negation

?

# 2. Stratification practice

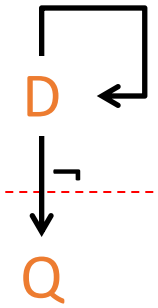
Parent(P,C)

Q: Find all descendants of Alice,
who are not descendants of Bob

```
D(x,y) :- Parent(x,y).
D(x,z) :- Parent(y,z), D(x,y).
Q(x) :- D('Alice',x), ¬D('Bob',x).
```

first compute for each person their descendants

then use negation



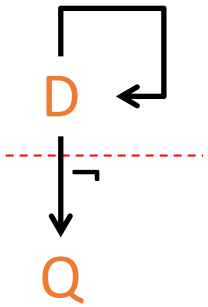
# 2. Stratification practice

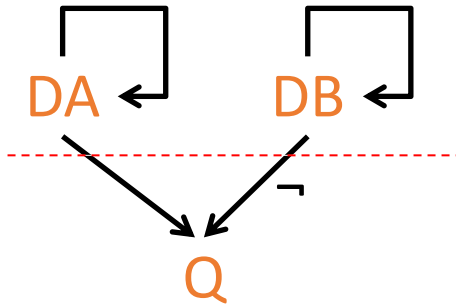
Parent(P,C)

Q: Find all descendants of Alice, who are not descendants of Bob

```
D(x,y) :- Parent(x,y).
D(x,z) :- Parent(y,z), D(x,y).
Q(x) :- D('Alice',x), ¬D('Bob',x).
```

```
DA(y) :- Parent('Alice',y).
DA(y) :- Parent(x,y), DA(x).
DB(y) :- Parent('Bob',y).
DB(y) :- Parent(x,y), DB(x).
Q(x) :- DA(x), ¬DB(x).
```

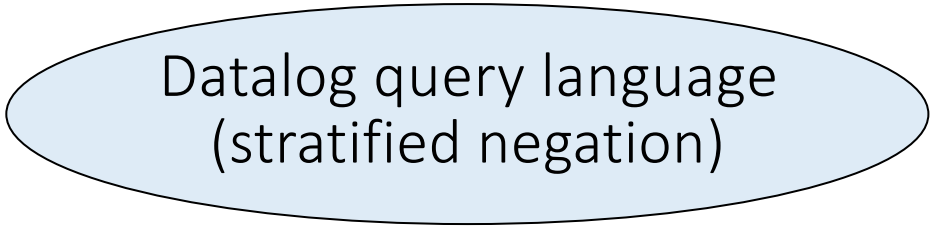


# Outline: T1-4: Datalog

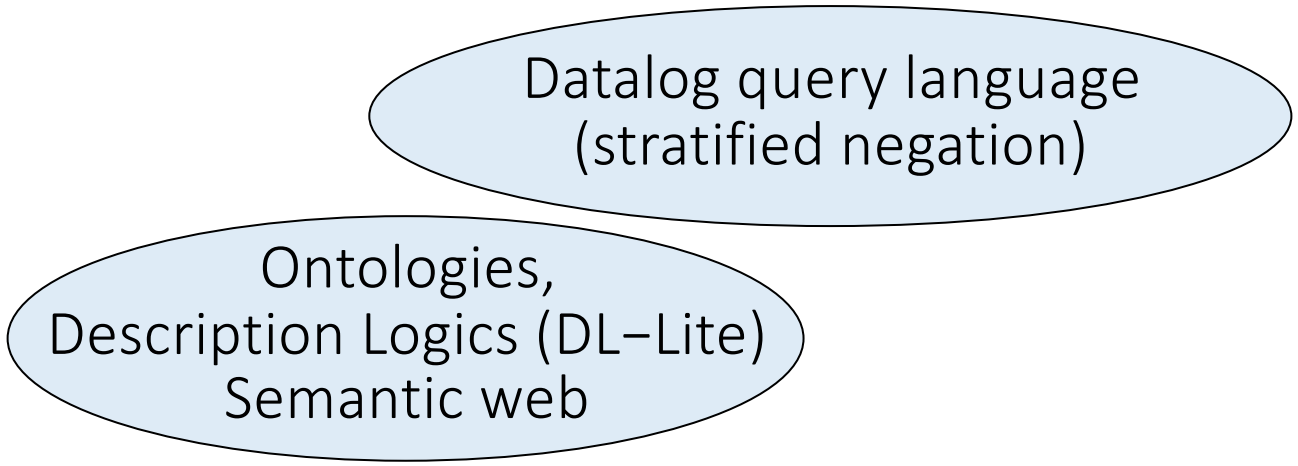
- Datalog
  - Datalog rules
  - Recursion
  - Recursion in SQL [moved here from T1-U1: SQL]
  - Semantics
  - Datalog$^{\neg}$: Negation, stratification
  - Datalog$^{\pm}$
  - Stable model semantics (Answer set programming)
  - Datalog vs. RA
  - Naive and Semi-naive evaluation (incl. Incremental View Maintenance)

# Datalog$^{\pm}$: background




- Much is possible with Datalog

# Datalog$^{\pm}$: background

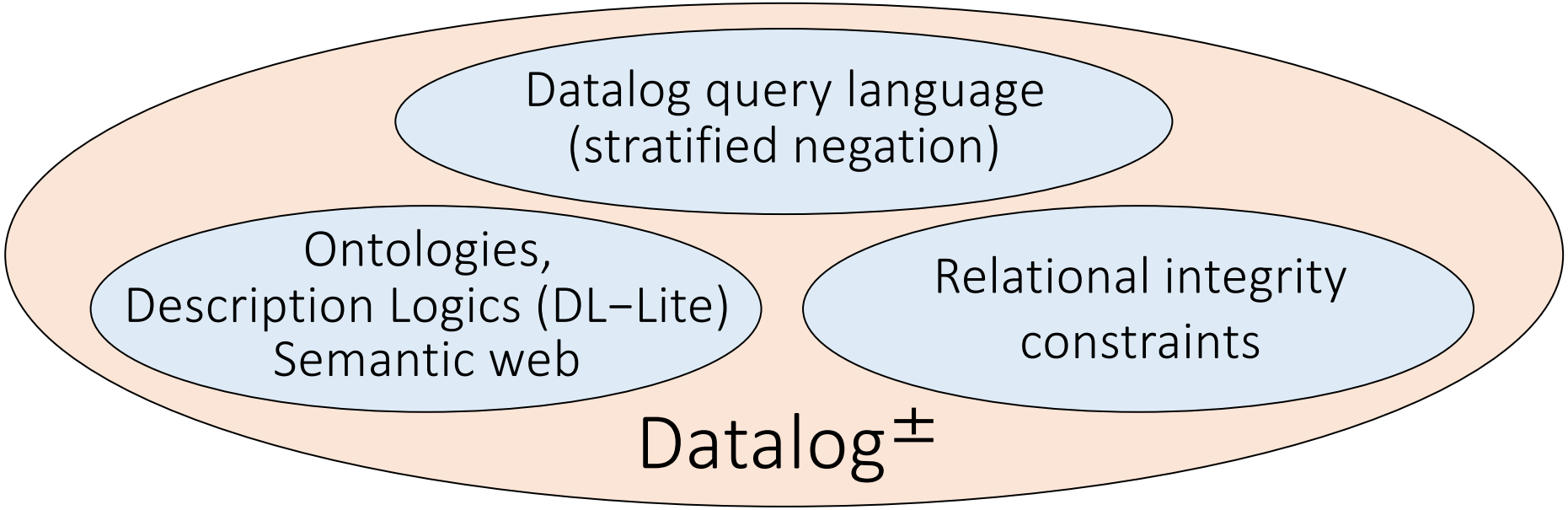



- Much is possible with Datalog
- Much is not (observed e.g. by [Patel-Schneider, Horrocks 2006])

Patel-Schneider, Horrocks. Position paper: A comparison of two modelling paradigms in the Semantic Web. WWW (Semantic Web track). 2006. https://dl.acm.org/doi/10.1145/1135777.1135784

Based on a presentation by Andrea Cali
Wolfgang Gatterbauer. Principles of scalable data management: https://northeastern-datalab.github.io/cs7240/

# Datalog$^{\pm}$: goal




- Much is possible with Datalog
- Much is not (observed e.g. by Patel-Schneider, Horrocks 2006)
- Datalog$^{\pm}$ is a framework that extends Datalog with:
  - value invention (∃-variables in the head): TGDs (Tuple-Generating Dependencies)
  - equality predicate in the head: EGDs (Equality Generating Dependencies)
  - constant ⊥ in the head: negative constraints (disjointness)

Patel-Schneider, Horrocks. Position paper: A comparison of two modelling paradigms in the Semantic Web. WWW (Semantic Web track). 2006. https://dl.acm.org/doi/10.1145/1135777.1135784
Cali, Gottlob, Lukasiewicz, Marnette, Pieris. Datalog+/-: A Family of Logical Knowledge Representation and Query Languages for New Applications. LICS 2010. https://doi.org/10.1109/LICS.2010.27
Based on a presentation by Andrea Cali
Wolfgang Gatterbauer. Principles of scalable data management: https://northeastern-datalab.github.io/cs7240/

# Datalog and expressiveness for ontological reasoning

| Assertion type | Datalog rule |
|---|---|
| Inclusion | $\text{emp}(X) \rightarrow \text{person}(X)$ |
| (Inverse) role inclusion | $\text{reportsTo}(X, Y) \rightarrow \text{manages}(Y, X)$ |
| Reflexive expansion | $\text{boss}(X) \rightarrow \text{manages}(X, X)$ |
| Transitivity | $\text{manages}(X, Y), \text{manages}(Y, Z) \rightarrow \text{manages}(X, Z)$ |
| Concept product | $\text{seniorEmp}(X), \text{emp}(Y) \rightarrow \text{higher}(X, Y)$ |
| Participation | ? |
| Disjointness | ? |
| Functionality | ? |

| Ontology assertion | Datalog$^{\pm}$ rule |
|---|---|
| Participation | $\text{boss}(X) \rightarrow \exists Y\ \text{reports}(Y, X)$ |
| Disjointness | $\text{customer}(X), \text{boss}(X) \rightarrow \bot$ |
| Functionality | $\text{reports}(X, Y1), \text{reports}(X, Y2) \rightarrow Y1 = Y2$ |

Based on a presentation by Andrea Cali
Wolfgang Gatterbauer. Principles of scalable data management: https://northeastern-datalab.github.io/cs7240/

# Datalog$^{\pm}$ vs. DL

$$
\begin{aligned}
&FiveStar(X) \rightarrow Hotel(X),\\
&FiveStar(X), \text{not}\, Pool(X,Y) \rightarrow \exists Z\ Beach(X,Z),\\
&FiveStar(X), \text{not}\, Beach(X,Y) \rightarrow \exists Z\ Pool(X,Z),\\
&Beach(X,Y) \rightarrow \exists Z\ SwimOpp(X,Z),\\
&Pool(X,Y) \rightarrow \exists Z\ SwimOpp(X,Z),
\end{aligned}
$$

The above example corresponds to the following set of DL axioms, expressed in an extension of $\mathcal{ELHI}$ by nonmonotonic negation:

$$
\begin{aligned}
FiveStar &\sqsubseteq \text{Hotel},\\
FiveStar \sqcap \text{not} \exists Pool &\sqsubseteq \exists Beach,\\
FiveStar \sqcap \text{not} \exists Beach &\sqsubseteq \exists Pool,\\
\exists Beach &\sqsubseteq \exists SwimOpp,\\
\exists Pool &\sqsubseteq \exists SwimOpp,
\end{aligned}
$$

Source: Gottlob, Lukasiewicz, Pieris. Datalog+/-: Questions and Answers. AAAI 2014. https://www.aaai.org/ocs/index.php/KR/KR14/paper/viewPaper/7965
Wolfgang Gatterbauer. Principles of scalable data management: https://northeastern-datalab.github.io/cs7240/

# Interesting Observations

- Exploiting schema knowledge in query answering is not trivial
- Languages and algorithms exist that allow for tractable query answering
- Applications in real-world scenarios are possible
  - Industrial applications in data integration, Semantic Web, ontological reasoning
  - Companies and Products: RelationalAI, Deepreason.ai, Oracle Semantic Technologies, IBM IODT, OntoDLV (Vienna)

# Outline: T1-4: Datalog

- Datalog
  - Datalog rules
  - Recursion
  - Recursion in SQL [moved here from T1-U1: SQL]
  - Semantics
  - Datalog$^{\neg}$: Negation, stratification
  - Datalog$^{\pm}$
  - Stable model semantics (Answer set programming)
  - Datalog vs. RA
  - Naive and Semi-naive evaluation (incl. Incremental View Maintenance)

# Negation in Souffle vs. Negation in ASP

## Negation in Rules

A rules of the form

```
CanRenovate(person, building) :- Owner(person, building), !Heritage(building).
```

expresses the rule that an owner can renovate a building with the condition that the building is not classified as heritage. Thus the literal "Heritage(building)" is negated (via "!") in the body of the rule. Not all negations are semantically permissible. For example,

```
A(x) :- ! B(x).
B(x) :- ! A(x).
```

is a circular definition. One cannot determine if anything belongs to the relation "A" without determining if it belongs to relation "B". But to determine if it is a "B" one needs to determine if the item belongs to "A". Such circular definitions are forbidden. Technically, rules involving negation must be stratifiable.

Negated literals do not bind variables. For example,

```
A(x,y) :- R(x), !S(y).
```

is not valid as the set of values that "y" can take is not clear. This can be rewritten as,

```
A(x,y) :- R(x), Scope(y), !S(y).
```

where the relation "Scope" defines the set of values that "y" can take.

YES: stable model semantics as used by ASP can deal with this circular definition

NO: but safety conditions are still the same as for souffle

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

# Answer Set Programming (ASP)

- Programming paradigm that can model AI problems (e.g, planning, combinatorics)
- Basic idea
  - Allow non-stratified negation and encode problem (specification & "instance") as logic program rules
  - Solutions are stable models of the program
- Semantics based on Possible Worlds and Stable Models
  - Given an answer set program P, there can be multiple solutions (stable models, answer sets)
  - Each model M: assignment of true/false value to propositions to make all formulas true (combinatorial)
  - Captures default reasoning, non-monotonic reasoning, constrained optimization, exceptions, weak exceptions, preferences, etc., in a natural way
- Finding stable models of answer set programs is not easy
  - Current systems CLASP, DLV, Smodels, etc., extremely sophisticated
  - Work by first grounding the program, suitably transforming it to a propositional theory whose models are stable models of the original program (contrast with "lifted inference" later )
  - These models are found using a SAT solver

# Rules with Negation

- Closed world assumption (CWA) as used in standard Datalog:
  - If a fact does not logically follow from a set of Datalog clauses, then we conclude that the negation of this fact is true.
- Problem: CWA can lead to inconsistencies when negation is allowed in rule bodies. Intuition: we can have multiple minimal models ("Herbrand models")

Example 1:

```
boring(chess) :- boring(chess).
```

⇨ ?

What are all the possible *minimal* models:

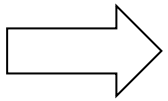
- Herbrand universe $U_P$ (set of all constants) ={chess}
- Herbrand base $B_P$ (set of grounded atoms) = {boring(chess)}
- Interpretations (all subsets of $B_P$) = { {}, {boring(chess)} }
- Model: interpretation that makes each ground instance of each rule true

# Rules with Negation

- Closed world assumption (CWA) as used in standard Datalog:
  - If a fact does not logically follow from a set of Datalog clauses, then we conclude that the negation of this fact is true.
- Problem: CWA can lead to inconsistencies when negation is allowed in rule bodies. Intuition: we can have multiple minimal models ("Herbrand models")

Example 1:

```
boring(chess) :- boring(chess).
```



$M_1 = \{\}$

What are all the possible *minimal* models:

$M_2$ = {boring(chess)} is a model, but not minimal

# Rules with Negation

- Closed world assumption (CWA) as used in standard Datalog:
  - If a fact does not logically follow from a set of Datalog clauses, then we conclude that the negation of this fact is true.
- Problem: CWA can lead to inconsistencies when negation is allowed in rule bodies. Intuition: we can have multiple minimal models ("Herbrand models")

Example 1:

```
boring(chess) :- boring(chess).
```

⇒ $M_1 = \{\}$

What are all the possible *minimal* models:

$M_2 = \{boring(chess)\}$ is a model, but not minimal

Example 2:

```
boring(chess) :- ¬interesting(chess).
```

⇒ ?

What are all the possible *minimal* models:

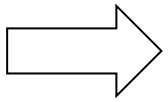
Possible interpretations:
{ {},
{b(c)}, {i(c)},
{b(c),i(c)} }

# Rules with Negation

- Closed world assumption (CWA) as used in standard Datalog:
  - If a fact does not logically follow from a set of Datalog clauses, then we conclude that the negation of this fact is true.
- Problem: CWA can lead to inconsistencies when negation is allowed in rule bodies. Intuition: we can have multiple minimal models ("Herbrand models")

Example 1:

```
boring(chess) :- boring(chess).
```



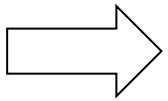
$M_1 = \{\}$

What are all the possible *minimal* models:

$M_2 = \{boring(chess)\}$ is a model, but not minimal

Example 2:

```
boring(chess) :- ¬interesting(chess).
```



$M_1 = \{boring(chess)\}$

$M_2 = \{interesting(chess)\}$

What are all the possible *minimal* models:

# Semantics: Informally

- Informally, a stable model M of a ground program P is a set of ground atoms such that
  1. Every rule is satisfied:

     i.e., for any rule in P

     $h$ :- $a_1$, ..., $a_m$, ¬$b_1$, ..., ¬$b_n$.

     if each atom $a_i$ is satisfied ($a_i$'s are in M) and no atom $b_i$ is satisfied (i.e. no $b_i$ is in M), then h is in M.
  2. Every $h \in M$ can be derived from a rule by a "non-circular reasoning" (informal for: we are looking for minimal models, or there is some "derivation provenance")

Recall that alternatives to "¬" are "not" and "!". Writing out "not" explicitly is more common in ASP.

# Semantics: "non-circular" more formally

Idea: Guess a model M (= a set of atoms). Then verify M is the exact set of atoms that "can be derived" under standard minimal model semantics on $P^M$ on a modified positive program $P^M$ (called "the reduct") derived from P as follows:

1. Create all possible groundings of the rules of program P
2. Delete all grounded rules that contradict M

   ~~$h :- a_1, ..., a_m, \neg b_1, ..., \neg b_n.$~~ if some $b_i \in M$

3. In remaining grounded rules, delete all negative literals

   $h :- a_1, ..., a_m, \neg b_1, ..., \neg b_n.$ if no $b_i \in M$

M is a stable model of P iff M is the least model of $P^M$

# Semantics: "non-circular" more concisely

The reduct of P w.r.t M is:

$P^M = \{$ h :- $a_1, ..., a_m$. | h :- $a_1, ..., a_m, \neg b_1, ..., \neg b_n$. $\in$ grounding of P $\wedge$ no $b_i \in M \}$

M is a stable model of P iff M is the least model of $P^M$