Logic in CS and Databases

• Logic has had an immense impact on CS
• Computing has strongly driven one particular branch of logic: finite model theory
  – That is, (FO/SO) logic restricted to finite models
  – Tight connections to complexity theory!
  – The basis of branches in Artificial Intelligence
• Natural tool to capture and attack fundamental problems in database management
  – Relations as first-class citizens
  – Inference for assuring data integrity
  – Inference for question answering (queries)
• Used for developing and analyzing the relational model from the early days (Codd, 1972)
Outline

• Crash course on First-Order Logic (FOL)
  • Relational Calculus
    ▪ Syntax and Semantics
    ▪ Domain Independence and Safety
    ▪ Equivalence to RA
  • Datalog
    ▪ Syntax and Semantics
    ▪ Recursion
    ▪ Negation
3 Components of FOL

1. FO language
   – *What are the allowed syntactic expressions?*
   – DB world: schema, constraints, query language

2. Interpretation
   – *Mapping symbols to an actual world*
   – DB world: database

3. Semantics
   – *When is a statement “true” under some interpretation?*
   – DB world: meaning of constraint satisfaction and query results
Components of FOL: (1) FO Language

- **Alphabet**: symbols in use
  - Variables, constants, **function** symbols, **predicate** symbols, connectives, quantifiers, punctuation symbols

- **Term**: expression that stands for an *element*
  - Variable, constant
  - Inductively $f(t_1,\ldots,t_n)$ where $t_i$ are terms, $f$ a function symbol

- **(Well-formed) formula**: parameterized statement
  - *Atom* $p(t_1,\ldots,t_n)$ where $p$ is a predicate symbol, $t_i$ terms
  - Inductively, for formulas $F$, $G$, variable $X$:
    \[
    F \land G \quad F \lor G \quad \neg F \quad F \rightarrow G \quad F \leftrightarrow G \quad \forall X F \quad \exists X F
    \]

- A **first-order language** refers to the set of all formulas over an alphabet
Components of FOL: (2) Interpretation

• An *interpretation* INT for an alphabet consists of:
  – A non-empty set $\text{Dom}$, called *domain*
  – An assignment of an element in $\text{Dom}$ to each constant symbol
  – An assignment of a function $\text{Dom}^n \rightarrow \text{Dom}$ to each $n$-ary function symbol
  – An assignment of a function $\text{Dom}^n \rightarrow \{\text{true, false}\}$ (i.e., a relation) to each $n$-ary predicate symbol
Components of FOL: (3) Semantics

- A **variable assignment** to a formula in an interpretation \( \text{INT} \) assigns to each **free variable** \( X \) a value from \( \text{Dom} \)
  - A **free variable** is one used without quantification, e.g., \( \text{Person}(X), \exists Y \text{Married}(X,Y) \) (more formal definition later)

- **Truth value** for formula \( F \) under interpretation \( \text{INT} \) and variable assignment \( V \):
  - Atom \( p(t_1,\ldots,t_n): q(s_1,\ldots,s_n) \) where \( q \) is the interpretation of the predicate \( p \) and \( s_i \) the interpretation of \( t_i \)
  - \( F \land G \quad F \lor G \quad \neg F \quad F \rightarrow G \quad F \leftrightarrow G \): according to truth table
  - \( \exists X F \): true iff there exists \( d \in \text{Dom} \) such that if \( V \) assigns \( d \) to \( X \) then the truth value of \( F \) is true; otherwise false
  - \( \forall X F \): true iff for all \( d \in \text{Dom} \), if \( V \) assigns \( d \) to \( X \) then the truth value of \( F \) is true; otherwise false

- If a formula has no free vars (closed formula), we can simply refer to its truth value under \( \text{INT} \)
Outline

• Crash course on First-Order Logic (FOL)

• Relational Calculus
  ▪ Syntax and Semantics
  ▪ Domain Independence and Safety
  ▪ Equivalence to RA

• Datalog
  ▪ Syntax and Semantics
  ▪ Recursion
  ▪ Negation
1. RC = FOL over DB
2. RC can express “bad queries” that depend not only on the DB, but also on the domain from which values are taken [domain dependence]
3. We cannot test whether an RC query is “good,” but we can use a ”good” subset of RC that captures all “good” queries [safety]
4. “Good” RC and RA can express the same queries! [equivalence]
Relational Calculus (RC)

• RC is, essentially, first-order logic (FO) over the schema relations
  – A query has the form “find all tuples \((x_1,\ldots,x_k)\) such that \(F(x_1,\ldots,x_k)\) holds”

• RC is a *declarative* query language
  – Meaning: a query is not defined by a sequence of operations, but rather by a condition that the result should satisfy
Outline

• Crash course on First-Order Logic (FOL)
• Relational Calculus
  ▪ Syntax and Semantics
  ▪ Domain Independence and Safety
  ▪ Equivalence to RA
• Datalog
  ▪ Syntax and Semantics
  ▪ Recursion
  ▪ Negation
Which relatives does this query find?
RC Symbols

• Constant values: a, b, c, ...
  – Values that may appear in table cells

• Variables: x, y, z, ...
  – Range over the values that may appear in table cells

• Relation symbols: R, S, T, Person, Parent, ...
  – Each with a specified arity
  – Will be fixed by the relational schema at hand
  – No attribute names, only attribute positions!

• Unlike general FOL, no function symbols!
Atomic RC Formulas

• Atomic formulas:
  – $R(t_1, \ldots, t_k)$
    • $R$ is a $k$-ary relation
    • Each $t_i$ is a variable or a constant
    • Semantically it states that $(t_1, \ldots, t_k)$ is a tuple in $R$
    • Example: $\text{Person}(x, \text{'female'}, \text{'Canada'})$
  – $x \ op \ u$
    • $x$ is a variable, $u$ is a variable/constant, $\text{op}$ is one of $>, <, =, \neq$
    • Simply binary predicates, predefined interpretation
    • Example: $x=y, z>5$
• Formula:
  – Atomic formula
  – If $\phi$ and $\psi$ are formulas then these are formulas:
    \[ \begin{align*}
      \phi \land \psi & \quad \phi \lor \psi & \quad \phi \rightarrow \psi & \quad \phi \rightarrow \psi & \quad \neg \phi & \quad \exists x \phi & \quad \forall x \phi
    \end{align*} \]

    \[ \begin{align*}
      \text{Person}(u, \text{'female'}, \text{'Canada'}) \land \\
      \exists y, z \left[ \text{Parent}(z, y) \land \text{Parent}(y, x) \land \\
      \exists w \left[ \text{Parent}(z, w) \land y \neq w \land (u = w \lor \text{Spouse}(u, w)) \right]\right]
    \end{align*} \]
Free Variables

• Variables not bound to quantifiers

• Formally:
  
  – A free variable of an atomic formula is a variable that occurs in the atomic formula
  
  – A free variable of $\varphi \land \psi, \varphi \lor \psi, \varphi \rightarrow \psi$ is a free variable of either $\varphi$ or $\psi$
  
  – A free variable of $\neg \varphi$ is a free variable of $\varphi$
  
  – A free variable of $\exists x \varphi$ and $\forall x \varphi$ is a free variable $y$ of $\varphi$ such that $y \neq x$

• We write $\varphi(x_1, \ldots, x_k)$ to states that $x_1, \ldots, x_k$ are the free variables of $\varphi$ (in some order)
What Are the Free Variables?

Person(\(u\), 'female', 'Canada') \land
\exists y, z \left[ \text{Parent}(z, y) \land \text{Parent}(y, x) \land \exists w \left[ \text{Parent}(z, w) \land y \neq w \land \right.ight.
\left( u = w \lor \text{Spouse}(u, w) \right) \left. \right) \right] \right] \]

Notation: \( \varphi(x, u) \), CandianAunt(\(u, x\)), …
Relation Calculus Query

- An **RC query** is an expression of the form
  \[
  \{ \langle x_1, \ldots, x_k \rangle \mid \varphi(x_1, \ldots, x_k) \}
  \]
  where \( \varphi(x_1, \ldots, x_k) \) is an RC formula

- An RC query is **over** a relational schema \( S \) if all the relation symbols belong to \( S \) (with matching arities)
RC Query

\{ (x,u) \mid \text{Person}(u, 'female', 'Canada') \land \\
\exists y,z [\text{Parent}(z,y) \land \text{Parent}(y,x) \land \\
\exists w [\text{Parent}(z,w) \land y \neq w \land \\
(u=w \lor \text{Spouse}(u,w))] ] } \}

- Person(id, gender, country)
- Parent(parent, child)
- Spouse(person1, person2)
### Who took all core courses?

$$\text{Studies} \div \pi_{\text{course}} \sigma_{\text{type}=\text{core}} \text{CourseType}$$
R=S in Primitive RA vs. RC

\[ R(X,Y) \div S(Y) \]

In RA:
\[ \pi_X R \setminus \pi_X \left( (\pi_X R \times S) \setminus R \right) \]

In RC:
\[ \{ (X) \mid \exists Z \left[ R(X,Z) \right] \land \forall Y \left[ S(Y) \rightarrow R(X,Y) \right] \} \]
• There are two common variants of RC:
  – **DRC**: *Domain Relational Calculus* (what we're doing)
  – **TRC**: *Tuple Relational Calculus*

• DRC applies vanilla FO: terms interpreted as *attribute values*, relations have *arity* but no *attribute names*

• TRC is more “database friendly”: terms interpreted as *tuples* with *named attributes*

• There are easy conversions between the two formalisms (nothing deep)
[Complimentary] Our Example in TRC

\{ t | \exists a [a \in \text{Person} \land a[\text{gender}] = 'female' \land a[\text{country}] = 'Canada'] \land \\
\exists p, q [p \in \text{Parent} \land p[\text{child}] = t[\text{nephew}] \land \\
q \in \text{Parent} \land q[\text{child}] = p[\text{parent}] \land \\
\exists w [w \in \text{Parent} \land w[\text{parent}] = q[\text{parent}] \land \\
w[\text{child}] \neq q[\text{child}] \land ((t[\text{aunt}] = w[\text{child}] \land t[\text{aunt}] = a[id]) \lor \exists s [s \in \text{Spouse} \land \\
s[\text{person1}] = w[\text{child}] \land s[\text{person2}] = t[\text{aunt}] \land t[\text{aunt}] = a[id]])}
Outline

• Crash course on First-Order Logic (FOL)
• Relational Calculus
  ▪ Syntax and Semantics
  ▪ Domain Independence and Safety
    ▪ Equivalence to RA
• Datalog
  ▪ Syntax and Semantics
  ▪ Recursion
  ▪ Negation
What is the Meaning of the Following?

\[
\{ (x) \mid \neg \text{Person}(x, 'female', 'Canada') \}
\]

\[
\{ (x,y) \mid \exists z [\text{Spouse}(x,z) \land y=z] \}
\]

\[
\{ (x,y) \mid \exists z [\text{Spouse}(x,z) \land y\neq z] \}
\]

- Person(id, gender, country)
- Parent(parent, child)
- Spouse(person1, person2)
Bringing in the Domain

• Let $S$ be a schema, let $D$ be a database over $S$, and let $Q$ be an RC query over $S$

• $D$ gives an interpretation for the underlying FOL
  – Predicates $\rightarrow$ relations; constants copied; no functions
  – … almost! What is the domain?

• The active domain (of $D$ and $Q$) is the set of all the values that occur in either $D$ or $Q$

• The query $Q$ is evaluated over $D$ with respect to a domain $\text{Dom}$ that contains the active domain

• Denote by $Q^{\text{Dom}}(D)$ the result of evaluating $Q$ over $D$ relative to the domain $\text{Dom}$
Domain Independence

• Let $\mathbf{S}$ be a schema, and let $Q$ be an RC query over $\mathbf{S}$

• We say that $Q$ is *domain independent* if for every database $D$ over $\mathbf{S}$ and every two domains $\text{Dom}_1$ and $\text{Dom}_2$ that contain the active domain, we have:

$$Q^{\text{Dom}_1}(D) = Q^{\text{Dom}_2}(D)$$
Which One is Domain Independent?

\{ (x) | \neg \text{Person}(x, 'female', 'Canada') \} \\
\{ (x,y) | \exists z \left[ \text{Spouse}(x,z) \land y=z \right] \} \\
\{ (x,y) | \exists z \left[ \text{Spouse}(x,z) \land y \neq z \right] \} \\
\{ (x) | \exists z,w \text{ Person}(x,z,w) \land \exists y \left[ \neg \text{Likes}(x,y) \right] \} \\
\{ (x) | \exists z,w \text{ Person}(x,z,w) \land \forall y \left[ \neg \text{Likes}(x,y) \right] \} \\
\{ (x) | \exists z,w \text{ Person}(x,z,w) \land \forall y \left[ \neg \text{Likes}(x,y) \right] \land \exists y \left[ \neg \text{Likes}(x,y) \right] \} \\

- \text{Person}(id, gender, country) \\
- \text{Likes}(person1, person2) \\
- \text{Spouse}(person1, person2)
Which One is Domain Independent?

\[
\begin{align*}
\{ & (x) \mid \neg \text{Person}(x, \text{'female'}, \text{'Canada'}) \} & & \text{Not DI} \\
\{ & (x, y) \mid \exists z \ [\text{Spouse}(x, z) \land y = z] \} & & \text{DI} \\
\{ & (x, y) \mid \exists z \ [\text{Spouse}(x, z) \land y \neq z] \} & & \text{DI} \\
\{ & (x) \mid \exists z, w \ \text{Person}(x, z, w) \land \forall y \ [\neg \text{Likes}(x, y)] \} & & \text{DI} \\
\{ & (x) \mid \exists z, w \ \text{Person}(x, z, w) \land \forall y \ [\neg \text{Likes}(x, y)] \} & & \text{DI} \\
\{ & (x) \mid \exists z, w \ \text{Person}(x, z, w) \land \forall y \ [\neg \text{Likes}(x, y)] \land \exists y \ [\neg \text{Likes}(x, y)] \} & & \text{DI}
\end{align*}
\]

- Person(id, gender, country)
- Likes(person1, person2)
- Spouse(person1, person2)
Bad News...

• We would like be able to tell whether a given RA query is domain independent,
  — ... and then reject “bad queries”

• Alas, this problem is undecidable!
  — That is, there is no algorithm that takes as input an RC query and returns true iff the query is domain independent
Domain-independent RC has an effective syntax; that is:

- A syntactic restriction of RC in which every query is domain independent
  - Restricted queries are said to be safe
- Safety can be tested automatically (and efficiently)
- Most importantly, for every domain independent RC query there exists an equivalent safe RC query!
Safety

• We do not formally define the safe syntax in this course
• Details on the safe syntax can be found in the textbook *Foundations of Databases* by Abiteboul, Hull and Vianu
  – Example:
    • In $\exists x \varphi$, the variable $x$ should be *guarded* by $\varphi$
    • Every variable is guarded by $R(x_1,\ldots,x_k)$
    • In $\varphi \land (x=y)$, the variable $x$ is guarded if and only if either $x$ or $y$ is guarded by $\varphi$
    • ... and so on
Outline

• Crash course on First-Order Logic (FOL)
• Relational Calculus
  ▪ Syntax and Semantics
  ▪ Domain Independence and Safety
  ▪ Equivalence to RA
• Datalog
  ▪ Syntax and Semantics
  ▪ Recursion
  ▪ Negation
Example Revisited

\[ R(X,Y) \div S(Y) \]

In RA:
\[ \pi_X R \setminus \pi_X \left( (\pi_R X \times S) \setminus R \right) \]

In RC:
\[ \{ (X) \mid \exists Z \left[ R(X,Z) \right] \land \forall Y \left[ S(Y) \rightarrow R(X,Y) \right] \} \]
Another Example

- Person(id, gender, country)
- Parent(parent, child)
- Spouse(person1, person2)

\[
\{ (x) \mid \exists z, w \text{ Person}(x, z, w) \land \forall y [\neg \text{Spouse}(x, y)] \}\n\]

\[
\pi_{id}\text{Person} \setminus \rho_{\text{person1/id}}\pi_{\text{person1}}\text{Spouse}
\]
**THEOREM:** RA and domain-independent RC have the same expressive power.

More formally, on every schema $S$:

- For every RA expression $E$ there is a domain-independent RC query $Q$ such that $Q \equiv E$
- For every* domain-independent RC query $Q$ there is an RA expression $E$ such that $Q \equiv E$

* Technicality: we consider only queries that output values from the database (otherwise we need to extend RA accordingly…*)
The proof has two directions:

1. Translate a given RA query into an equivalent RC query
2. Translate a given RC query into an equivalent RA query

Part 1 is fairly easy: induction on the size of the RA expression

Part 2 is more involved
Intuition on RA $\rightarrow$ RC

- Construction by induction
- Technicality: need to maintain a mapping between attribute names and variables (simple)

<table>
<thead>
<tr>
<th>RA</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R$ (n columns)</td>
<td>$R(X_1,\ldots,X_n)$</td>
</tr>
<tr>
<td>$E_1 \times E_2$</td>
<td>$F_1 \land F_2$ disjoint variables (rename)</td>
</tr>
<tr>
<td>$E_1 - E_2$</td>
<td>$F_1 \land \neg F_2$ use identical variables (rename)</td>
</tr>
<tr>
<td>$E_1 \cup E_2$</td>
<td>$F_1 \lor F_2$ use identical variables (rename)</td>
</tr>
<tr>
<td>$\Pi_{a_1,\ldots,a_k}(E_1)$</td>
<td>$\exists X_1 \ldots \exists X_m F_1$ where $X_1,\ldots,X_m$ are the variables not among $a_1,\ldots,a_k$</td>
</tr>
<tr>
<td>$\sigma_c(E_1)$</td>
<td>$F_1 \land c$</td>
</tr>
</tbody>
</table>

Here, $F_i$ is the formula constructed for $E_i$
1. RC = FOL over DB

2. RC can express “bad queries” that depend not only on the DB, but also on the domain from which values are taken [domain dependence]

3. We cannot test whether an RC query is “good,” but we can use a ”good” subset of RC that captures all “good” queries [safety]

4. “Good” RC and RA can express the same queries! [equivalence]
• Crash course on First-Order Logic (FOL)
• Relational Calculus
  ▪ Syntax and Semantics
  ▪ Domain Independence and Safety
  ▪ Equivalence to RA

• Datalog
  ▪ Syntax and Semantics
  ▪ Recursion
  ▪ Negation
Datalog

- Database query language
- “Clean” restriction of Prolog with DB access
  - Expressive & declarative:
    - *Set-of-rules* semantics
    - Independence of execution order
    - Invariance under logical equivalence
- Mostly academic implementations; some commercial instantiations

Path(x, y) ← Edge(x, y)
Path(x, z) ← Edge(x, y), Path(y, z)
InCycle(x) ← Path(x, x)
Outline

• Crash course on First-Order Logic (FOL)
• Relational Calculus
  ▪ Syntax and Semantics
  ▪ Domain Independence and Safety
  ▪ Equivalence to RA
• Datalog
  ▪ Syntax and Semantics
  ▪ Recursion
  ▪ Negation
Example

- Person(id, gender, country)
- Parent(parent, child)
- Spouse(person1, person2)

Invited(y) ← Relative('myself', y), Local(y)
Local(x) ← Person(x, y, 'IL')
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)

Sometimes :- used instead of ←, e.g.,

Local(x) :- Person(x, y, 'IL')
EDBs and IDBs

- Datalog rules operate over:
  - **Extensional Database (EDB)** predicates
    - These are the provided/stored database relations from the relational schema
  - **Intentional Database (IDB)** predicates
    - These are the relations *derived* from the stored relations through the rules
Example

- Person(id, gender, country)
- Parent(parent, child)
- Spouse(person1, person2)

Local(x) $\Leftarrow$ Person(x, y, 'IL')
Relative(x, x) $\Leftarrow$ Person(x, y, z)
Relative(x, y) $\Leftarrow$ Relative(x, z), Parent(z, y)
Relative(x, y) $\Leftarrow$ Relative(x, z), Parent(y, z)
Relative(x, y) $\Leftarrow$ Relative(x, z), Spouse(z, y)
Invited(y) $\Leftarrow$ Relative('myself', y), Local(y)

EDB
IDB
Datalog Program

- An *atomic formula* has the form $R(t_1,\ldots,t_k)$ where:
  - $R$ is a $k$-ary relation symbol
  - Each $t_i$ is either a constant or a variable
- A *Datalog rule* has the form
  \[
  \text{head} \leftarrow \text{body}
  \]
  where *head* is an atomic formula and *body* is a sequence of atomic formulas
  - For simplicity, we disallow constants in the head
- A *Datalog program* is a set of Datalog rules
Logical Interpretation of a Rule

• Consider a Datalog rule of the form

\[ R(x) \leftarrow \psi_1(x,y), \ldots, \psi_m(x,y) \]

– Here, \( x \) and \( y \) are disjoint sequences of variables, and each \( \psi_i(x,y) \) is an atomic formula with variables among \( x \) and \( y \)

– Example: \( \text{TwoPath}(x_1,x_2) \leftarrow \text{Edge}(x_1,y), \text{Edge}(y,x_2) \)

• The rule stands for the logical formula

\[ \forall x \left[ R(x) \leftarrow \exists y [\psi_1(x,y) \land \cdots \land \psi_m(x,y)] \right] \]

– Example: if there exists \( y \) where \( \text{Edge}(x_1,y) \) and \( \text{Edge}(y,x_2) \) hold, then \( \text{TwoPath}(x_1,x_2) \) should hold
Syntactic Constraints

- We require the following from the rule
  \[ R(x) \leftarrow \psi_1(x, y), \ldots, \psi_m(x, y) \]

  1. Safety: every variable in \( x \) should occur in the body at least once
  2. The predicate \( R \) must be an IDB predicate
     - (The body can include both EDBs and IDBs)

- Example of forbidden rules:
  - \( R(x, z) \leftarrow S(x, y), R(y, x) \)
  - \( \text{Edge}(x, y) \leftarrow \text{Edge}(x, z), \text{Edge}(x, y) \)
    - Assuming Edge is EDB
Semantics of Datalog Programs

• Let $S$ be a schema, $D$ a database over $S$, and $P$ be a Datalog program over $S$
  – That is, all EDBs predicates belong to $S$

• The result of evaluating $P$ over $D$ is a database $I$ over the IDB schema of $P$

• We give several definitions:
A *chase* procedure is a program of the following form:

\[
\text{Chase}(P,D)
\]

- \( I := \text{empty} \)
- \( \textbf{while}(\text{true}) \) {
  - if(\(D \cup I\) satisfies all the rules of \(P\))
    - return \(I\)
  - Find a rule \(\text{head}(x) \leftarrow \text{body}(x,y)\) and tuples \(a, b\) such that \(D \cup I\) contains \(\text{body}(a,b)\) but not \(\text{head}(a)\)
  - \(I := I \cup \{\text{head}(a)\}\)
}
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*
Example

- Person(id, gender, country)
- Parent(parent, child)
- Spouse(person1, person2)

Relative(x,x) $\iff$ Person(x,y,z)
Relative(x,y) $\iff$ Relative(x,z), Parent(z,y)
Invited(y) $\iff$ Relative('1',y)

<table>
<thead>
<tr>
<th>Person</th>
<th>Parent</th>
<th>Relative</th>
<th>Invited</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>F</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>M</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>M</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Person(id, gender, country)
- Parent(parent, child)
- Spouse(person1, person2)

Relative(x,x) $\iff$ Person(x,y,z)
Relative(x,y) $\iff$ Relative(x,z), Parent(z,y)
Invited(y) $\iff$ Relative('1',y)
### Example

- **Person** (id, gender, country)
- **Parent** (parent, child)
- **Spouse** (person1, person2)

**Relative**

- `Relative(x,x) <-> Person(x,y,z)`
- `Relative(x,y) <-> Relative(x,z), Parent(z,y)`
- `Invited(y) <-> Relative('1',y)`

**Tables**

<table>
<thead>
<tr>
<th>Person</th>
<th>Parent</th>
<th>Relative</th>
<th>Invited</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

**Person**

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>gender</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>F</td>
<td>IL</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>M</td>
<td>AU</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>NL</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>M</td>
<td>IL</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>FR</td>
<td></td>
</tr>
</tbody>
</table>

**Parent**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

**Relative**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Spouse**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>


Example

- Person(id, gender, country)
- Parent(parent, child)
- Spouse(person1, person2)

Relative(x,x) \iff\ Person(x,y,z)
Relative(x,y) \iff\ Relative(x,z), Parent(z,y)
Invited(y) \iff\ Relative('1',y)

<table>
<thead>
<tr>
<th>Person</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Spouse</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relative</th>
<th>Invited</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Example

- Person(id, gender, country)
- Parent(parent, child)
- Spouse(person1, person2)

Relative(x, x) ← Person(x, y, z)
Relative(x, y) ← Relative(x, z), Parent(z, y)
Invited(y) ← Relative('1', y)

<table>
<thead>
<tr>
<th>Person</th>
<th>1</th>
<th>F</th>
<th>IL</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>M</td>
<td>AU</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>NL</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>M</td>
<td>IL</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>FR</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parent</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relative</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Spouse</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Invited</th>
<th>1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

Relative(x,x) ← Person(x,y,z)
Relative(x,y) ← Relative(x,z), Parent(z,y)
Invited(y) ← Relative('1',y)

<table>
<thead>
<tr>
<th>Person</th>
<th>Parent</th>
<th>Relative</th>
<th>Invited</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Person</th>
<th>Parent</th>
<th>Relative</th>
<th>Invited</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Person(id, gender, country)
- Parent(parent, child)
- Spouse(person1, person2)
### Example

- Person(id, gender, country)
- Parent(parent, child)
- Spouse(person1, person2)

\[
\text{Relative}(x,x) \iff \text{Person}(x,y,z)
\]
\[
\text{Relative}(x,y) \iff \text{Relative}(x,z), \text{Parent}(z,y)
\]
\[
\text{Invited}(y) \iff \text{Relative}('1',y)
\]

<table>
<thead>
<tr>
<th></th>
<th>Person</th>
<th>Parent</th>
<th>Relative</th>
<th>Invited</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>id</td>
<td>gender</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>IL</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>M</td>
<td>AU</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>NL</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>M</td>
<td>IL</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>FR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Notes
- The `Relative` relation is used to store relationships between individuals.
- The `Parent` relation connects parents and children.
- The `Spouse` relation links two individuals as a couple.
- The `Invited` relation indicates if an individual was invited to an event.
Example

- Person(id, gender, country)
- Parent(parent, child)
- Spouse(person1, person2)

Relative(x,x) \leftrightarrow \text{Person}(x,y,z)
Relative(x,y) \leftrightarrow \text{Relative}(x,z), \text{Parent}(z,y)
Invited(y) \leftrightarrow \text{Relative}('1',y)
Model-Theoretic Definition

- We say that $I$ is a model of $P$ (w.r.t. $D$) if $D \cup I$ satisfies all the rules of $P$.
- We say that $I$ is a minimal model is $I$ does not properly contain any other model.
**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
3. Pick a chase and prove by induction: If $I'$ is a model, then every intermediate $I$ is contained in $I'$

The minimal model is the *result*, denoted $P(D)$.
• We can associate with each rule an RA expression, for example:
  – path(X, Y) \texttt{\leftarrow} E(X, Y) with E
  – path(X, Y) \texttt{\leftarrow} E(X, Z), \ Path(Z, Y) with \ \pi_{1,4} (E \bowtie_{2=3} Path)

• The \textit{RA expansion} for predicate Path, denoted exp[Path], is the union:
  \ \ E \cup \pi_{1,4} (E \bowtie_{2=3} Path)

• (Implicitly rename all attributes as needed)
Another definition: minimal fixpoint

– Here, a fixpoint means a database I such that for all IDB predicates R we have
I.R=exp[R](D∪I)
Outline

• Crash course on First-Order Logic (FOL)
• Relational Calculus
  ▪ Syntax and Semantics
  ▪ Domain Independence and Safety
  ▪ Equivalence to RA
• Datalog
  ▪ Syntax and Semantics
  ▪ Recursion
  ▪ Negation
“Recursive” Program?

Local(x) ← Person(x,y,'IL')
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)

Local(x) ← Person(x,y,'IL')
Relative(x,x) ← Person(x,y,z)
Invited(y) ← Relative('myself',y), Local(y)

MayLike(x,y) ← Close(x,z), Likes(z,y)
Visit(x,y) ← MayLike(x,y)
Close(x,z) ← Visit(x,y), Visit(z,y)
• 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 edge \(R \rightarrow S\) whenever there is a rule with \(S\) in the head and \(R\) in the body
• A Datalog program is recursive if its dependency graph contains a cycle
Recursive?

\[ \text{Local}(x) \leftarrow \text{Person}(x,y,'IL') \]
\[ \text{Relative}(x,x) \leftarrow \text{Person}(x,y,z) \]
\[ \text{Relative}(x,y) \leftarrow \text{Relative}(x,z), \text{Parent}(z,y) \]
\[ \text{Relative}(x,y) \leftarrow \text{Relative}(x,z), \text{Parent}(y,z) \]
\[ \text{Relative}(x,y) \leftarrow \text{Relative}(x,z), \text{Spouse}(z,y) \]
\[ \text{Invited}(y) \leftarrow \text{Relative}('myself',y), \text{Local}(y) \]
Local\( (x) \leftarrow \text{Person}(x, y, 'IL') \)

Relative\( (x, x) \leftarrow \text{Person}(x, y, z) \)

Invited\( (y) \leftarrow \text{Relative}('myself', y), \text{Local}(y) \)
And This One?

\[
\begin{align*}
\text{MayLike}(x,y) & \leftarrow \text{Close}(x,z), \text{Likes}(z,y) \\
\text{Visit}(x,y) & \leftarrow \text{MayLike}(x,y) \\
\text{Close}(x,z) & \leftarrow \text{Visit}(x,y), \text{Visit}(z,y)
\end{align*}
\]
THEOREM: Datalog can express queries that RA (RC) cannot
(example: transitive closure of a graph)

Proof not covered in the course
THEOREM: Non-recursive Datalog has the same expressive power as the algebra \( \{\sigma_\equiv, \pi, \times, \rho, \cup\} \)
(where \( \sigma_\equiv \) means selection with a single equality)

Proof: exercise (not covered)

This fragment is often called “positive RA” or USPJ (union-select-project-join)
Can Datalog express difference?

Answer: No!

Proof: Datalog is monotone
- That is, if $D$ and $D'$ are such that every relation of $D$ is contained in the corresponding relation of $D'$, then $P(D) \subseteq P(D')$
Outline

• Crash course on First-Order Logic (FOL)
• Relational Calculus
  ▪ Syntax and Semantics
  ▪ Domain Independence and Safety
  ▪ Equivalence to RA
• Datalog
  ▪ Syntax and Semantics
  ▪ Recursion
  ▪ Negation
What is the Semantics?

Adding negation to Datalog is not straightforward!

Buddy\((x,y)\) ⟷ Likes\((x,y), \neg \text{Parent}(y,x)\)

Buddy\((x,y)\) ⟷ \neg \text{Buddy}(x,y), \neg \text{Anti}(x,y), \text{Likes}(x,y)

Anti\((x,y)\) ⟷ \neg \text{Buddy}(x,y), \text{Suspects}(x,y)

Likes('Avi','Alma')
Suspects('Avi','Alma')

Buddy('Avi','Alma')

Anti('Avi','Alma')
Negation in Datalog

• Various semantics have been proposed for supporting negation in Datalog
  – In a way that makes sense

• We will look at two:
  – *Semipositive* programs (restricted)
  – *Stratified* programs (standard)
A *semipositive* program is a program where only EDBs may be negated

- Safety: every variable occurs in a positive literal
  - Guarantees domain independence
- Semantics: same as ordinary Datalog programs

\[
\text{Buddy}(x, y) \leftarrow \text{Likes}(x, y), \neg \text{Parent}(y, x)
\]
Stratified Programs

• Let $P$ be a Datalog program
• Let $E_0$ be set of EDB predicates
• A *stratification* of $P$ is a partitioning of the IDB predicates into disjoint sets $E_1, \ldots, E_k$ where:
  – For $i=1, \ldots, k$, every rule with head in $E_i$ has body predicates only from $E_0, \ldots, E_i$
  – For $i=1, \ldots, k$, every rule with head in $E_i$ can have negated body predicates only from $E_0, \ldots, E_{i-1}$
Example

- Person(id, gender, country)
- Fake(id)
- Parent(parent, child)
- Spouse(person1, person2)
- Likes(person1, person2)

\[
\begin{align*}
\text{RealPerson}(x) & \iff \text{Person}(x, y, z), \neg \text{Fake}(x) \\
\text{Relative}(x, x) & \iff \text{RealPerson}(x) \\
\text{Relative}(x, y) & \iff \text{Relative}(x, z), \text{Parent}(z, y) \\
\text{Relative}(x, y) & \iff \text{Relative}(x, z), \text{Parent}(y, z) \\
\text{Relative}(x, y) & \iff \text{Relative}(x, z), \text{Spouse}(z, y) \\
\text{Buddy}(x, y) & \iff \neg \text{Relative}(x, y), \text{Likes}(x, y) \\
\text{Buddy}(x, y) & \iff \neg \text{Relative}(x, y), \text{Buddy}(x, z), \text{Buddy}(z, y)
\end{align*}
\]
Another Stratification?

- Person(id, gender, country)
- Fake(id)
- Parent(parent, child)
- Spouse(person1, person2)
- Likes(person1, person2)

\[
\begin{align*}
\text{RealPerson}(x) & \leftarrow \text{Person}(x,y,z), \neg \text{Fake}(x) \\
\text{Relative}(x,x) & \leftarrow \text{RealPerson}(x) \\
\text{Relative}(x,y) & \leftarrow \text{Relative}(x,z), \text{Parent}(z,y) \\
\text{Relative}(x,y) & \leftarrow \text{Relative}(x,z), \text{Parent}(y,z) \\
\text{Relative}(x,y) & \leftarrow \text{Relative}(x,z), \text{Spouse}(z,y) \\
\text{Buddy}(x,y) & \leftarrow \neg \text{Relative}(x,y), \text{Likes}(x,y) \\
\text{Buddy}(x,y) & \leftarrow \neg \text{Relative}(x,y), \text{Buddy}(x,z), \text{Buddy}(z,y)
\end{align*}
\]
Semantics of Stratified Programs

• For $i=1,...,k$:
  – Compute the IDBs of the stratum $E_i$
  – Add computed IDBs to the EDBs

• Then, due to the definition of stratification, each $E_i$ can be viewed as **semipositive**

• Does the result depend on the specific stratification of choice?
  – Answer on the next slide
Theorems on Stratification (1)

- **Theorem 1**: All stratifications are equivalent
  - That is, they give the same result on every input

- **Theorem 2**: 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
Theorems on Stratification (2)

• **Theorem 3:** Non-recursive Datalog programs with negation are stratifiable
  – Via the topological order

• **Theorem 4:** Nonrecursive Datalog with negation has the same expressive power as the algebra \{\sigma_-, \pi, \times, \rho, \cup, \setminus\}
  – Extendable to RA if we add the predicates >, <