Relational Algebra is the core language for databases, but its expressibility is limited. The most basic problem that is not expressible in RA is the graph transitive closure. In the following lectures, we will introduce a new language, called Datalog, that allows us to express more complex problems by adding recursion. Datalog has seen many applications over the last years, including data integration, declarative networking, and program analysis.

### 8.1 Datalog Syntax

A Datalog rule is an expression of the form

\[ R(\vec{x}) : R_1(\vec{x}_1), R_2(\vec{x}_2), \ldots, R_n(\vec{x}_n) \]  

This is the same syntax as the one we used for Conjunctive Queries; the big difference is that we now allow for a relation \( R \) to appear both on the left and the right side of the rule. A Datalog program is defined as a finite set of Datalog rules.

In Datalog we have two different types of schemas:

- **Extensional Schema**: this consists of extensional relations (EDBs), which occur only in the right-hand-side of the rules. Such relations are intuitively the "input" of the Datalog program.

- **Intensional Schema**: this consists of intensional relations (IDBs), which occur at least once in the left-hand-side of a rule. Intensional relations are the "output" of the Datalog program.

A Datalog program semantically is a mapping from instances over the extensional schema to instances over the intensional schema. Let’s see some example of Datalog programs below.

#### Example 1

Let \( R(A, B) \) be a relation that contains the edges of a directed graph. The following Datalog program computes the transitive closure (TC) of the graph. Recall that TC computes all the pairs \((u, v)\) of vertices, such that there is a directed path from node \(u\) to node \(v\):

\[ T(x, y) : R(x, y). \]
\[ T(x, y) : T(x, z), R(z, y). \]

The second rule is called a linear rule, because the intensional relation \( T \) of the head appears
exactly once in the right-hand-side. The following Datalog program, which also computes transitive closure, contains a non-linear rule.

\[
T(x, y) := R(x, y).
\]
\[
T(x, y) := T(x, z), T(z, y).
\]

**Example 2**

Let us again assume that \(R(A, B)\) describes the edges of a directed graph. We want to write a Datalog program that computes (i) the nodes of the graph such that there exists a cycle of odd length that goes through, (ii) the nodes of the graph such that there exists a cycle of even length that goes through, and (iii) the nodes of the graph such that there exists a cycle of any length that goes through:

\[
\text{OddPath}(x, y) := R(x, y).
\]
\[
\text{EvenPath}(x, y) := R(x, z), \text{OddPath}(z, y).
\]
\[
\text{OddPath}(x, y) := R(x, z), \text{EvenPath}(z, y).
\]
\[
\text{OddCycle}(x) := \text{OddPath}(x, x).
\]
\[
\text{EvenCycle}(x) := \text{EvenPath}(x, x).
\]
\[
\text{Cycle}(x) := \text{OddCycle}(x).
\]
\[
\text{Cycle}(x) := \text{EvenCycle}(x).
\]

The relations \(\text{OddPath}\) and \(\text{EvenPath}\) are called mutually recursive relations, because their definition depends on each other.

### 8.2 Datalog Semantics

There are three different equivalent semantics for Datalog: model-theoretic, fixpoint and proof-theoretic. Here we will discuss only the first two.

#### 8.2.1 Model-Theoretic Semantics

We start by associating a (first-order) logical sentence to each Datalog rule. For example, the rule \(\rho : T(x, y) :- T(x, z), R(z, y)\) gives the following logical sentence: \(\phi_{\rho} = \forall x, y, z (T(x, z) \land R(z, y) \rightarrow T(x, y))\). In general, for a rule \(\rho\) of the form (8.1), we associate the following logical sentence:

\[
\phi_{\rho} = \forall x_1, \ldots, x_k (R_1() \land R_2() \cdots \land R_k() \rightarrow R())
\]

where \(x_1, \ldots, x_k\) are the variables in the body of the rule. An interesting observation is that the logical sentences of the above form are *Horn clauses*: Horn clauses are formulas that consist of a disjunction of literals, where there exists at most one positive literal.
Let \( \Sigma_P \) be the set of logical sentences \( \phi_\rho \), for every rule \( \rho \) in the Datalog program \( P \).

**Definition 1**

Let \( P \) be a Datalog program. A pair of instances \((I, J)\), where \( I \) is an EDB, and \( J \) is an IDB, is a model of \( P \) if \((I, J)\) satisfies \( \Sigma_P \).

Given an EDB \( I \), the minimal model of \( P \), denoted \( J = P(I) \), is a minimal IDB \( J \) such that \((I, J)\) is a model of \( P \).

We can show that a minimal model always exists, and it is also unique. Also, the minimal model contains only tuples with values from the active domain \( \text{adom}(I) \). The semantics of a Datalog program \( P \) executed on EDB \( I \) is exactly the minimum model \( P(I) \).

**Exercise 1**

Consider the transitive closure on the following instance: \( I = \{ R(1,2) \, R(2,3), R(3,4) \} \). What is the minimal model in this case? Can you find a non-minimal model and a non-model?

### 8.2.2 Fixpoint Semantics

Let \( P \) be a Datalog program, and an instance \( J \) over the intensional and extensional schema. We say that a fact/tuple \( t \) is an immediate consequence of \( J \) if either \( t \in J \), or it is the direct result of a rule application using the instance \( J \). The immediate consequence operator for \( P \), denoted \( T_P \), maps an instance to another instance (over the intensional and extensional schema), such that \( T_P(J) \) contains all the facts that are immediate consequences of instance \( J \). It is easy to see that by our definition, \( T_P(J) \supseteq J \). Also, the operator is monotone:

**Proposition 1**

If \( I \subseteq J \) then \( T_P(I) \subseteq T_P(J) \).

**Definition 2**

We say that an instance \( I \) over the schema is a fixpoint for \( T_P \) if \( T_P(I) = I \).

We can now show the connection of the fixpoint semantics to the model-theoretic semantics.

**Theorem 1**

For each Datalog program \( P \) and EDB \( I \), the immediate consequence operator \( T_P \) has a unique, minimal fixpoint \( J \supseteq I \), which equals the model \( P(I) \).

The fixpoint semantics give us an algorithm that computes the output of a Datalog program. We start with the input \( I \), which is an EDB instance. We then compute \( T_P(I) \), then \( T_P(T_P(I)) \), and so on. Recall that the operator \( T_P \) is monotone. Also, at every iteration we compute at least one
new immediate consequence, and there is only a polynomial number of such tuples (since any new tuple must use values from the active domain). Thus, after a polynomial number of steps, we must reach a fixpoint. This way of evaluating Datalog is called the *naive* evaluation strategy.

**Exercise 2**

Consider the transitive closure on the following instance: \( I = \{ R(1,2), R(2,3), R(3,4) \} \). Show the application of the operator \( T_P \) until it reaches the fixpoint.

### 8.3 More on Datalog

**Proposition 2**

Every Datalog program \( P \) is monotone.

There are many interesting properties one can express in Datalog.

**Example 3**

Suppose we have a relation \( A(x, y) \) that expresses the fact that \( y \) is the parent of \( x \). The following Datalog program, called *same generation*, computes the pair \((u, v)\) that have a common ancestor and belong in the same "generation" w.r.t. to the ancestor.

\[
S(u, v) \leftarrow A(u, x), A(v, x).
\]
\[
S(u, v) \leftarrow A(u, x), S(x, y), A(v, y).
\]

A more modern application of Datalog is in program analysis, and in particular points-to analysis [PT]. In this setting, we are given a program (in any programming language), and we want to compute what points to what.

**Example 4**

This example describes the Datalog rules for a simple type of points-to analysis in C programs, called Andersen’s analysis. Initially, we turn instructions in C to predicates in Datalog:

- \( y=\&x \) : AddressOf\((y, x)\)
- \( y=x \) : Assign\((y, x)\)
- \( y=*x \) : Load\((y, x)\)
- \( *y=x \) : Store\((y, x)\)

We want to compute the relation PointsTo\((y,x)\), i.e. whether variable \( y \) may point to the location of variable \( x \). We can do this using the following Datalog program:
PointsTo(y,x) :- AddressOf(y,x).
PointsTo(y,x) :- Assign(y,z), PointsTo(z,x).
PointsTo(y,w) :- Load(y,x), PointsTo(x,z), PointsTo(z,w).
PointsTo(z,w) :- Store(y,x), PointsTo(y,z), PointsTo(x,w).

References
