Introduction to Datalog

Datalog introduces **recursion** into database queries

- Use deterministic rules to derive new information from given facts
- Inspired by logic programming (Prolog)
- However, no function symbols and no negation
- Studied in AI (knowledge representation) and in databases (query language)

**Example 12.1:** Transitive closure $C$ of a binary relation $r$

$$
C(x, y) \leftarrow r(x, y)
$$

$$
C(x, z) \leftarrow C(x, y) \land r(y, z)
$$

Intuition:
- some facts of the form $r(x, y)$ are given as input, and the rules derive new conclusions $C(x, y)$
- variables range over all possible values (implicit universal quantifier)

Syntax of Datalog

**Recall:** A **term** is a constant or a variable. An **atom** is a formula of the form $R(t_1, \ldots, t_n)$ with $R$ a predicate symbol (or relation) of arity $n$, and $t_1, \ldots, t_n$ terms.

**Definition 12.2:** A **Datalog rule** is an expression of the form:

$$
H \leftarrow B_1 \land \ldots \land B_m
$$

where $H$ and $B_1, \ldots, B_m$ are atoms. $H$ is called the **head** or **conclusion**; $B_1 \land \ldots \land B_m$ is called the **body** or **premise**. A rule with empty body ($m = 0$) is called a **fact**.

A ground rule is one without variables (i.e., all terms are constants).

A set of Datalog rules is a **Datalog program**.
### Datalog: Example

father(alice, bob) 
mother(alice, carla) 
mother(evan, carla) 
father(carla, david) 

\[ \text{Parents}(x, y) \leftarrow \text{father}(x, y) \]
\[ \text{Parents}(x, y) \leftarrow \text{mother}(x, y) \]
\[ \text{Ancestor}(x, z) \leftarrow \text{Parents}(x, y) \wedge \text{Ancestor}(y, z) \]
\[ \text{SameGeneration}(x, y) \leftarrow \text{Parents}(x, y) \wedge \text{Parent}(y, w) \wedge \text{SameGeneration}(w, v) \]

### Notes:
- **n** = 0 for ground facts, so they can always be derived (induction base)
- If variables in the head do not occur in the body, they can be any constant from the universe

### Datalog Semantics by Deduction (2)

**Definition 12.4:** Consider a Datalog program \( P \). The set of ground atoms that can be derived from \( P \) is the smallest set of atoms \( A \) for which there is a rule
\[ H \leftarrow B_1 \land \ldots \land B_n \]
and a ground substitution \( \theta \) such that

- \( A = H\theta \), and
- For each \( i \in \{1, \ldots, n\} \), \( B_i\theta \) can be derived from \( P \).

**Informally:**
- Restrict to set of constants that occur in program (finite)
- Variables can represent arbitrary constants from this set
- Ground substitutions map variables to constants
- A rule can be applied if its body is satisfied for some ground substitution

**Example 12.3:** The rule \( \text{Parents}(x, y) \leftarrow \text{mother}(x, y) \) can be applied to \( \text{mother}(alice, carla) \) under substitution \( \{x \mapsto alice, y \mapsto carla\} \).

**Notes:**
- If a rule is applicable under some ground substitution, then the according instance of the rule head is entailed.

### Datalog Deductions as Proof Trees

We can think of deductions as tree structures:

- **Ancestor(alice, david)**
  \[ \text{Parent(alice, carla), Ancestor(carla, david), Parent(carla, david)} \]
  \[ (1) \text{father(alice, bob)} \]
  \[ (2) \text{mother(alice, carla)} \]
  \[ (3) \text{mother(evan, carla)} \]
  \[ (4) \text{father(carla, david)} \]
  \[ (5) \text{Parents}(x, y) \leftarrow \text{father}(x, y) \]
  \[ (6) \text{Parents}(x, y) \leftarrow \text{mother}(x, y) \]
  \[ (7) \text{Ancestor}(x, y) \leftarrow \text{Parents}(x, y) \]
  \[ (8) \text{Ancestor}(x, z) \leftarrow \text{Parents}(x, y) \wedge \text{Ancestor}(y, z) \]
- **Parent(alice, carla)**
  \[ (1) \text{father(alice, bob)} \]
  \[ (2) \text{mother(alice, carla)} \]
  \[ (3) \text{mother(evan, carla)} \]
  \[ (4) \text{father(carla, david)} \]
  \[ (5) \text{Parents}(x, y) \leftarrow \text{father}(x, y) \]
  \[ (6) \text{Parents}(x, y) \leftarrow \text{mother}(x, y) \]
  \[ (7) \text{Ancestor}(x, y) \leftarrow \text{Parents}(x, y) \]
  \[ (8) \text{Ancestor}(x, z) \leftarrow \text{Parents}(x, y) \wedge \text{Ancestor}(y, z) \]
Instead of using substitutions, we can also ground programs:

**Definition 12.5:** The *grounding* \(\text{ground}(P)\) of a Datalog program \(P\) is the set of all ground rules that can be obtained from rules in \(P\) by uniformly replacing variables with constants from the universe.

Derivations are described by the immediate consequence operator \(T_P\) that maps sets of ground facts \(I\) to sets of ground facts \(T_P(I)\):

- \(T_P(I) = \{ H \mid H \leftarrow B_1 \land \ldots \land B_n \in \text{ground}(P)\text{ and }B_1, \ldots, B_n \in I\}\)
- Least fixed point of \(T_P\): smallest set \(L\) such that \(T_P(L) = L\)
- Bottom-up computation: \(T^0_P = \emptyset\text{ and }T^{i+1}_P = T_P(T^i_P)\)
- The least fixed point of \(T_P\) is \(T^*_P = \bigcup_{i\geq 0} T^i_P\) (exercise)

**Observation:** Ground atom \(A\) is derived from \(P\) if and only if \(A \in T^*_P\)

---

**Example 12.6:** The rule

\[
\text{Ancestor}(x, z) \leftarrow \text{Parent}(x, y) \land \text{Ancestor}(y, z)
\]

corresponds to the implication

\[
\forall x, y, z. \text{Parent}(x, y) \land \text{Ancestor}(y, z) \rightarrow \text{Ancestor}(x, z).
\]

A set of FO implications may have many models
~ consider **least model** over the domain defined by the universe

**Theorem 12.7:** A fact is entailed by the least model of a Datalog program if and only if it can be derived from the Datalog program.

---

**Datalog Semantics: Overview**

There are three equivalent ways of defining Datalog semantics:

- Proof-theoretic: What can be proven deductively?
- Operational: What can be computed bottom up?
- Model-theoretic: What is true in the least model?

In each case, we restrict to the universe of given constants.
~ similar to active domain semantics in databases

---

**Datalog as a Query Language**

How can we use Datalog to query databases?
~ View database as set of ground facts
~ Specify which predicate yields the query result

**Definition 12.8:** A Datalog query is a pair \((R, P)\), where \(P\) is a Datalog program and \(R\) is the answer predicate.

The result of the query is the set of \(R\)-facts entailed by \(P\).

Datalog queries distinguish “given” relations from “derived” ones:

- predicates that occur in a head of \(P\) are **intensional database (IDB) predicates**
- predicates that only occur in bodies are **extensional database (EDB) predicates**

**Requirement:** database relations used as EDB predicates only
Datalog as a Generalisation of CQs

A conjunctive query \( \exists y_1, \ldots, y_m. A_1 \land \ldots \land A_l \) with answer variables \( x_1, \ldots, x_n \) can be expressed as a Datalog query \( (\text{Ans}, P) \) where \( P \) has the single rule:

\[
\text{Ans}(x_1, \ldots, x_n) \leftarrow A_1 \land \ldots \land A_l
\]

Unions of CQs can also be expressed (how?)

**Intuition:** Datalog generalises UCQs by adding recursion.

---

Proof

**Theorem 12.10:** UCQs have the same expressivity as non-recursive Datalog.

Proof: “Non-recursive Datalog can express UCQs”: Just discussed.

“UCQs can express non-recursive Datalog”: Obtained by resolution:

1. Given rules \( p_1 : R(s_1, \ldots, s_k) \leftarrow C_1 \land \ldots \land C_l \) and \( p_2 : H \leftarrow B_1 \land \ldots \land R(t_1, \ldots, t_n) \land \ldots \land B_m \)

   (w.l.o.g. having no variables in common with \( p_1 \))

2. such that \( R(t_1, \ldots, t_n) \) and \( R(s_1, \ldots, s_k) \) unify with most general unifier \( \sigma \),

3. the resolvent of \( p_1 \) and \( p_2 \) with respect to \( \sigma \) is

   \( H \sigma \leftarrow B_1 \sigma \land \ldots \land C_1 \sigma \land \ldots \land C_l \sigma \land \ldots \land B_m \sigma \).

**Unfolding of \( R \)** means to simultaneously resolve all occurrences of \( R \) in bodies of any rule, in all possible ways. After adding all these resolvents, we can delete all rules that contain \( R \) in body or head (assuming that \( R \) is not the answer predicate).

Now given a non-recursive Datalog program, unfold each non-answer predicate (in any order), \( \sim \) program with only the answer predicate in heads (requires non-reversiveness).

This is easy to express as UCQ (using equality to handle constants in heads). \( \square \)

---

Datalog and UCQs

We can make the relationship of Datalog and UCQs more precise:

**Definition 12.9:** For a Datalog program \( P \):

- An IDB predicate \( R \) depends on an IDB predicate \( S \) if \( P \) contains a rule with \( R \) in the head and \( S \) in the body.
- \( P \) is non-recursive if there is no cyclic dependency.

**Theorem 12.10:** UCQs have the same expressivity as non-recursive Datalog.

That is: a query mapping can be expressed by some UCQ if and only if it can be expressed by a non-recursive Datalog program.

However, Datalog can be exponentially more succinct (shorter), as illustrated in an exercise.

---

Datalog and Domain Independence

Domain independence was considered useful for FO queries

\( \sim \) results should not change if domain changes

Several solutions:

- Active domain semantics: restrict to elements mentioned in database or query
- Domain-independent queries: restrict to query where domain does not matter
- Safe-range queries: decidable special case of domain independence

Our definition of Datalog uses the active domain (=Herbrand universe) to ensure domain independence.
Safe Datalog Queries

Similar to safe-range FO queries, there are also simple syntactic conditions that ensure domain independence for Datalog:

**Definition 12.11:** A Datalog rule is **safe** if all variables in its head also occur in its body. A Datalog program/query is safe if all of its rules are.

Simple observations:
- safe Datalog queries are domain independent
- every Datalog query can be expressed as a safe Datalog query . . .
- . . . and unsafe queries are not much more succinct either (exercise)

Some texts require Datalog queries to be safe in general but in most contexts there is no real need for this

Complexity of Datalog

How hard is answering Datalog queries?

Recall:
- Combined complexity: based on query and database
- Data complexity: based on database; query fixed
- Query complexity: based on query; database fixed

Plan:
- First show upper bounds (outline efficient algorithm)
- Then establish matching lower bounds (reduce hard problems)

A Simpler Problem: Ground Programs

Let’s start with Datalog without variables

~ sets of ground rules a.k.a. propositional Horn logic program

Naive computation of $T^i_P$:

```
01 $T^0_P := \emptyset$
02 $i := 0$
03 repeat:
  04 $T^{i+1}_P := \emptyset$
  05 for $H \leftarrow B_1 \land \ldots \land B_k \in P$:
  06 if $\{B_1, \ldots, B_k\} \subseteq T^i_P$:
    07 $T^{i+1}_P := T^i_P \cup \{H\}$
  08 $i := i + 1$
09 until $T^{i+1}_P = T^i_P$
10 return $T^i_P$
```

How long does this take?
- At most $|P|$ facts can be derived
- Algorithm terminates with $i \leq |P| + 1$
- In each iteration, we check each rule once (linear), and compare its body to $T^i_P$ (quadratic)

~ polynomial runtime
Complexity of Propositional Horn Logic

Much better algorithms exist:

**Theorem 12.12 (Dowling & Gallier, 1984):** For a propositional Horn logic program $P$, the set $T^m_P$ can be computed in linear time.

Nevertheless, the problem is not trivial:

**Theorem 12.13:** For a propositional Horn logic program $P$ and a proposition (or ground atom) $A$, deciding if $A \in T^m_P$ is a P-complete problem.

Remark:
all P problems can be reduced to propositional Horn logic entailment yet not all problems in P (or even in NL) can be solved in linear time!

Datalog Complexity

A straightforward approach:

1. Compute the grounding $\text{ground}(P)$ of $P$ w.r.t. the database $I$
2. Compute $T^m_{\text{ground}(P)}$

Complexity estimation:
- The number of constants $N$ for grounding is linear in $P$ and $I$
- A rule with $m$ distinct variables has $N^m$ ground instances
- Step (1) creates at most $|P| \cdot N^M$ ground rules, where $M$ is the maximal number of variables in any rule in $P$
  - $\text{ground}(P)$ is polynomial in the size of $I$
  - $\text{ground}(P)$ is exponential in $P$
- Step (2) can be executed in linear time in the size of $\text{ground}(P)$

Summing up: the algorithm runs in $\text{P data complexity}$ and in $\text{ExpTime query and combined complexity}$

P-Hardness of Data Complexity

We need to reduce a P-hard problem to Datalog query answering

We restrict to a simple form of propositional Horn logic:
- facts have the usual form $H \leftarrow$
- all other rules have the form $H \leftarrow B_1 \land B_2$

Deciding fact entailment is still P-hard (exercise)

We can store such programs in a database:
- For each fact $H \leftarrow$, the database has a tuple $\text{Fact}(H)$
- For each rule $H \leftarrow B_1 \land B_2$, the database has a tuple $\text{Rule}(H, B_1, B_2)$
P-Hardness of Data Complexity (2)

The following Datalog program acts as an interpreter for propositional Horn logic programs:

\[
\begin{align*}
&\text{True}(x) \leftarrow \text{Fact}(x) \\
&\text{True}(x) \leftarrow \text{Rule}(x, y, z) \land \text{True}(y) \land \text{True}(z)
\end{align*}
\]

Easy observations:
- True(X) is derived if and only if \(X\) is a consequence of the original propositional program.
- The encoding of propositional programs as databases can be computed in logarithmic space.
- The Datalog program is the same for all propositional programs.

\(\leadsto\) Datalog query answering is \(\text{P}\)-hard for data complexity

Preparing for a Long Computation

We need to encode \(2^N\) time points and tape positions
\(\leadsto\) use binary numbers with \(N^k\) digits

So \(X\) and \(Y\) in atoms like head\((X, Y)\) are really lists of variables \(X = x_1, \ldots, x_N\) and \(Y = y_1, \ldots, y_N\), and the arity of head is \(2 \cdot N^k\).

\(\text{TODO:}\) define predicates that capture the order of \(N^k\)-digit binary numbers

For each number \(i \in [1, \ldots, N^k]\), we use predicates:
- \(\text{suc}\(i\)(X, Y): X + 1 = Y\), where \(X\) and \(Y\) are \(i\)-digit numbers
- \(\text{first}\(i\)(X): X is the \(i\)-digit encoding of 0
- \(\text{last}\(i\)(X): X is the \(i\)-digit encoding of \(2^i - 1\)

Finally, we can define the actual order for \(i = N^k\)
- \(\preceq\) (X, Y): X \(\preceq\) Y, where X and Y are \(i\)-digit numbers

ExpTime-Hardness of Query Complexity

A direct proof:
Encode the computation of a deterministic Turing machine for up to exponentially many steps

Recall that \(\text{ExpTime} = \bigcup_{k \in \mathbb{N}} \text{Time}(2^k)\)
- in our case, \(n = N\) is the number of database constants
- \(k\) is some constant

\(\leadsto\) we need to simulate up to \(2^N\) steps (and tape cells)

Main ingredients of the encoding:
- \(\text{state}\(n\)(X): the TM is in state \(q\) after \(X\) steps
- \(\text{head}\(i\)(X, Y): the TM head is at tape position \(Y\) after \(X\) steps
- \(\text{symbol}\(i\)(X, Y): the tape cell at position \(Y\) holds symbol \(v\) after \(X\) steps

\(\leadsto\) How to encode \(2^N\) time points X and tape positions Y?

Defining a Long Chain

We can define \(\text{suc}\(i\)(X, Y), \text{first}\(i\)(X),\) and \(\text{last}\(i\)(X)\) as follows:

\[
\begin{align*}
\text{suc}\(0\)(0, 1) & \leftarrow \text{suc}\(0\)(0, 1) \\
\text{suc}\(0\)(0, 0, Y) & \leftarrow \text{succ}\(0\)(X, Y) \\
\text{suc}\(1\)(1, X, 1, Y) & \leftarrow \text{suc}\(1\)(X, Y) \\
\text{suc}\(0\)(0, X, 1, Y) & \leftarrow \text{last}\(0\)(X) \land \text{first}\(1\)(Y) \\
\text{first}\(1\)(0, X) & \leftarrow \text{first}\(1\)(X) \\
\text{last}\(1\)(1, X) & \leftarrow \text{last}\(1\)(X)
\end{align*}
\]

for \(X = x_1, \ldots, x_i\)
and \(Y = y_1, \ldots, y_i\)
lists of \(i\) variables

Now for \(M = N^k\), we define \(\preceq\(M\)(X, Y)\) as the reflexive, transitive closure of \(\text{suc}\(M\)(X, Y)\):

\[
\begin{align*}
\preceq\(M\)(X, X) & \leftarrow \preceq\(M\)(X, X) \\
\preceq\(M\)(X, Z) & \leftarrow \preceq\(M\)(X, Y) \land \text{suc}\(M\)(Y, Z)
\end{align*}
\]
Initialising the Computation

We can now encode the initial configuration of the Turing Machine for an input word $a_1 \ldots a_n \in (\{ \mid \} \cup \{ \ldots \})^*$. We write $B_i$ for the binary encoding of a number $i$ with $M = N^k$ digits. We write $B_i$ for the binary encoding of a number $i$ with $M = N^k$ digits.

$$\begin{align*}
\text{state}_T(B_0) & \quad \text{where } q_0 \text{ is the TM's initial state} \\
\text{head}(B_0, B_0) & \\
\text{symbol}_r(B_0, B_i) & \quad \text{for all } i \in \{1, \ldots, n\} \\
\text{symbol}_l(B_0, Y) & \quad \text{where } Y = y_1, \ldots, y_M \\
\end{align*}$$

TM Transition and Acceptance Rules

For each transition $(q, q', q', d) \in \Delta$, we add rules:

$$\begin{align*}
symbol_r(X', Y) & \leftarrow \text{succ}^M(X, X') \land \text{head}(X, Y) \land \text{symbol}_l(X) \\
\text{state}_T(X') & \leftarrow \text{succ}^M(X, X') \land \text{head}(X, Y) \land \text{symbol}_l(X) \land \text{state}_T(X) \\
\end{align*}$$

Similar rules are used for inferring the new head position (depending on $d$)

Further rules ensure the preservation of unaltered tape cells:

$$\begin{align*}
symbol_r(X', Y) & \leftarrow \text{succ}^M(X, X') \land \text{symbol}_l(X, Y) \land \\
\text{head}(X, Z) & \land \text{succ}^M(Z, Z') \land \leq^M(Z', Y) \\
symbol_l(X', Y) & \leftarrow \text{succ}^M(X, X') \land \text{symbol}_l(X, Y) \land \\
\text{head}(X, Z) & \land \text{succ}^M(Z', Z) \land \leq^M(Y, Z') \\
\end{align*}$$

The TM accepts if it ever reaches the accepting state $q_{acc}$:

$$\text{accept}() \leftarrow \text{state}_{q_{acc}}(X)$$

Hardness Results

**Lemma 12.15:** A deterministic TM accepts an input in $\text{Time}(2^{O(n)})$ if and only if the Datalog program defined above entails the fact accept().

We obtain ExpTime-hardness of Datalog query answering:

- The decision problem of any language in $\text{ExpTime}$ can be solved by a deterministic TM in $\text{Time}(2^{O(n)})$ for some constant $k$
- In particular, there are ExpTime-hard languages $L$ with suitable deterministic TM $M$ and constant $k$
- For any input word $w$, we can reduce acceptance of $w$ by $M$ in $\text{Time}(2^{O(n)})$ to entailment of $\text{accept}()$ by a Datalog program $P(w, M, k)$
- $P(w, M, k)$ is polynomial in $k$ and the size of $M$ and $w$
  (in fact, it can be constructed in logarithmic space)

ExpTime-Hardness: Notes

Some further remarks on our construction:

- The constructed program does not use EDB predicates $\rightarrow$ database can be empty
- Therefore, hardness extends to query complexity
- Using a fixed (very small) database, we could have avoided the use of constants
- We used IDB predicates of unbounded arity $\rightarrow$ they are essential for the claimed hardness
Summary and Outlook

Datalog can overcome some of the limitations of first-order queries

Non-recursive Datalog can express UCQs

Datalog is more complex than FO query answering:
  - ExpTime-complete for query and combined complexity
  - P-complete for data complexity

Open questions:
  - Expressivity of Datalog
  - Query containment for Datalog
  - Implementation techniques for Datalog