A Chapter Coming to the End Soon

2.1 Datalog and SQL: Learning by Doing
2.2 Datalog: Syntax and (Basic) Semantics
2.3 Datalog, SQL, and Background

2.2 ending today (hopefully)
2.3 starting next Monday
2.3 ending next Wednesday
• A **deductive database** in Datalog is a set of facts and rules
  (later on in this lecture, we will reconsider this "definition" a bit).

• By now people are used to assume that each relation in a Datalog-DB is either
  defined by stored facts only (base relation) or defined by rules only (derived relation):
  "Standardization Assumption"

• If you want to extend a rule-defined relation with some facts (for expressing special
  cases), however, an **auxiliary relation** summarizing the special cases is necessary again:

\[
\begin{align*}
  p(a). \\
p(b). \\
p(X) & \leftarrow q(X,Y). \\
p(X) & \leftarrow s(X), t(X).
\end{align*}
\]

\[
\begin{align*}
  p_1(a). \\
p_1(b). \\
p(X) & \leftarrow p_1(X). \\
p(X) & \leftarrow q(X,Y). \\
p(X) & \leftarrow s(X), t(X).
\end{align*}
\]

**Not admissible!**

**Standardized**
"Closed World Assumption" (CWA)

- In databases up till now, one does not store negative information, but positive data only (or data derivable from stored data).

- Nevertheless, many (not all!) queries containing negation can be answered, e.g.:
  
  Which cities are no major cities?

- This is possible, because we (tacitly) assume that all facts in DB-relations which are not stored are wrong in reality – and, of course, that all stored facts are true!

- This assumption is called the "Closed World Assumption" (CWA) in DDB-literature:
  - Each piece of knowledge about "the world" (i.e., the resp. application area) is represented in the DB in form of positive facts (stored or derivable).
  - There is no doubt about true and false information (2-valued logic).
  - In an "open world" it would be necessary to distinguish between negative and unknown information (e.g. by storing false facts explicitly in addition to true ones).

Obviously, CWA is hopelessly idealistic – but there is hardly any alternative!
CWA: 1) All true facts are represented in the DB.
2) All facts in the DB are true.
3) All false facts form the complement (in set-theoretic terms) of the DB and, thus, exist implicitly only.

Reference set for constructing the complement:

Set of all syntactically constructable facts

Constructable from
• all relation names in the schema
• all constant in all value domains

Complement(DB)

The complement of the DB is never explicitly computed or even stored (for efficiency reasons)!
• Consequence of CWA: Negation in Datalog is admissible in rule bodies only, not in facts and not in the head of a rule (i.e., not for derivable facts).

• There is no directly expressible negative information in a Datalog-DB:

  not capital_of('Germany', 'Bonn') .

  No stored "negative facts" !!

• Derivation of negative information is excluded, too:

  not is_a_capital( X ) ← situated_in( X, 'Bavaria' ) .

  No derivable "negative facts" !!
• Deriving positive information exclusively from negative information is not possible in Datalog, too, because the complement of the DB would have to be made explicit for this purpose (which is unrealistic, see above):

\[
\text{north\_german\_city}(X) \leftarrow \text{not is\_situated\_in}(X, 'Bavaria').
\]

Negation is "unproductive"!!

• Negation in Datalog is admissible in combination with positive facts only, i.e., only in connection with logical conjunction: \text{and} \text{not}

• Negation can be used for "testing" variable bindings only, which have been "produced" in the positive parts of the rule before:

\[
\text{north\_german\_city}(X) \leftarrow \text{city}(X, Y1, Y2, Y3), \text{not is\_situated\_in}(X, 'Bavaria').
\]

Production Test
Evaluating Negative Literals

\[
\text{normal_city('Köln', 'K').}
\]

\[
\text{normal_city(City, Car) ← city(City, _, Car, _), not is_a_capital(City, Car).}
\]

\[
\]

\[
\text{is_a_capital('Köln', 'K')? yes!}
\]

\[
\text{is_a_capital('München', 'M'). is_a_capital('Düsseldorf', 'D'). ...}
\]

\[
\text{is_a_capital('Köln', 'K')? no!}
\]

\[
\text{Positive „side evaluation"}
\]
The corresponding evaluation principle for negative literals is called "Negation as failure". (If we fail to find the positive literal to be tested in the DB, we assume it to be false.)

**Prerequisite:**
- Before evaluating any negative literals, all variables contained in this literal have to be bound by evaluating "suitable" positive literals.
- Only negative ground literals are evaluable via "negation as failure".

In order to evaluate a literal \( \text{not } F \), . . .
- . . . try to answer its positive part \( F \).
- If \( F \) is true, then \( \text{not } F \) is false.
- If \( F \) is false, then \( \text{not } F \) is true:
  "(Proof of the) negation (of 'F') by failure (to prove 'F')"

Negative literals with variables are not evaluable by accessing DB-facts:
- 'not p(X)': Find X-bindings, so that 'p(X)' is not true (in the DB)!
- Inspecting the p-Relation produces only such X-bindings, for which 'p(X)' is true!
- Where to find "all the other possible" X-bindings? (Complement remains implicit due to CWA!)
• Thus, we need an additional safety condition for negated literals:

> Each variable occurring in a negative literal has to occur in at least one positive literal, too.

• **Dangerous**: Erroneous application of unsafe variables may easily happen, if misinterpreting the assumption about implicit existential quantification.

Would be intuitively meaningful, but doesn’t belong here according to the „quantifier rule“!
If the "forbidden" form of existential quantification is to be expressed in a different way, it is necessary to "swap" the existential quantifier into a separate rule:

```
∃ State:

normal_city(City, Car) ←
  city(City, _, Car, _),
  not capital_of(State, City).
```

```
∃ State:

∃ State:

normal_city(City, Car) ←
  city(City, _, Car, _),
  not is_a_capital(City).
```

```
∃ State:

is_a_capital(City) ←
capital_of(State, City).
```
Negation is admitted only if occurring **directly in front of individual literals**, not for negating entire conjunctive expressions. Thus, rule bodies **may not contain nesting** of logical operators!

```
north_or_west_german_city(X) ←
  city(X, _, _, _),
  not (south_of(X, 'Hannover'), east_of(X, 'Lübeck')).
```

If such a rule is to be expressed differently in Datalog, it is necessary to introduce another **auxiliary relation** defined by a separate rule (without nesting):

```
north_or_west_german_city(X) ←
  city(X, _, _, _),
  not south_east_of(X).

south_east_of(X) ←
  south_of(X, 'Hannover'),
  east_of(X, 'Lübeck').
```
Simulating Universal Quantifiers (1)

- We learnt previously, that all local variables in a rule body are implicitly existentially quantified. **What about universal quantifiers** („forall“ in logic – symbolically: ∀)?

- As a **motivating example**, consider the following natural language sentence (to be turned into a Datalog rule):

  A student is successful if (s)he has passed exams of all mandatory modules.

- For the corresponding Datalog formalization, consider relations `students(MatrNr)`, `exams(MatrNr,ModNr,Result)`, and `modules(ModNr,Status)`. Further assume that exam results are either **pass** or **fail**, and that module status is either **m(andatory)** or **optional**.

- There is **no forall in Datalog**, not even implicitly (as for **SQL**, also lacking FORALL)!

- Instead, a law of equivalence from predicate logic has to be exploited for „simulating“ **forall** by means of **not** and **exists**:

  \[
  \forall x: F(x) \equiv \neg \exists x: \neg F(x)
  \]

- In natural language, applying the same reasoning principle leads to the reformulation of the example as follows:

  A student is successful if **there is no** mandatory module (s)he did **not** pass.
A student is successful if there is no mandatory module (s)he did not pass.

• If Datalog would permit explicit quantifiers (and nesting), the following rule could be written, formalizing the above sentence:

\[
\text{successful}(S) \leftarrow \text{students}(S) \text{ and not } (\exists M: \text{modules}(M,'m') \text{ and not exams}(S,M,'pass')).
\]

• Beware! This is not Datalog, but hypothetical „extended Datalog“, we just use didactically!

• Applying techniques introduced before (unnesting, implicit existential quantification), we obtain the following (equivalent) version – using an auxiliary rule – which is proper Datalog:

\[
\begin{align*}
\text{successful}(S) & \leftarrow \\
& \text{students}(S), \text{not has\_failed\_module}(S). \\
\text{has\_failed\_module}(S) & \leftarrow \\
& \text{students}(S), \text{modules}(M,'m'), \text{not exams}(S,M,'pass')).
\end{align*}
\]
• Doesn’t belong to the „core“ of Datalog, by unavoidable in practice:

Comparison operators

• In logic: Relation names, too
  (like names of DB-relations)

• But: These „relations“ are obviously not definable by facts oder rules in extensional form, but have to be realized in external programming languages by means of suitable "test procedures" (i.e., from the perspective of Datalog as "built-ins")

• For better distinction between (DB-)relations and this kind of test relations we use another notion from logic (more or less synonym with „relation“): "Predicate"

• In Datalog, we use test predicates in test literals, e.g.

\[
X > Y \quad \quad X <= 1 \quad \quad \text{not} \quad a = b
\]
Safe Comparisons

• Comparison predicates are used exclusively for testing whether two elements of a certain data type denoted by two terms satisfy the resp. test.

• Comparisons are possible only if none of the two parameters of a test literal are still variable when performing the test.

• A test literal thus is subject to similar safety requirements as negative literals:

  Each variable in a test literal has to occur in at least one, positive DB-literal within the same conjunction which contains the resp. test literal.

• Examples for safe resp. unsafe usage of comparisons:

  Safe
  \[ p(X,Z), X > Y, q(Z,Y) \]

  Unsafe
  \[ p(X,Y), Y < Z \]
Built in-Functions

- Unavoidable, too: **Arithmetic operations**
  (and possibly other elementary operations on data types)

- Such "built in"-functions are to be realized in an external programming language, too.

- Evaluable (functional) terms in DB-literals are "disturbing" as they have to be treated different from the "matching"-based evaluation of DB-literals over facts and rules:

  \[ p(X, X+1) \leftarrow q(X, X-1). \]

  \[ p(X,Y) \leftarrow Y = X+1, q(X,Z), Z = X-1. \]

  Not really tests anymore!

  \[ \Rightarrow \text{Functional terms are (for now) admissible in test literals only!} \]
Aggregate Functions (1)

- important class of „built-in“ functions in SQL:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT</td>
<td>Number of</td>
</tr>
<tr>
<td>SUM</td>
<td>Sum</td>
</tr>
<tr>
<td>AVG</td>
<td>Average</td>
</tr>
<tr>
<td>MAX</td>
<td>Maximum</td>
</tr>
<tr>
<td>MIN</td>
<td>Minimum</td>
</tr>
</tbody>
</table>

Aggregate functions compute one scalar value out of a set of scalar values (the „aggregate“) originating from one column of one table:
Aggregates in SQL: Reminder

This is how aggregates work in SQL:

```
SELECT P.Rank, AVG(P.Age) AS AvgAge
FROM professors AS P
WHERE P.Name <> 'Ken'
GROUP BY P.Rank
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Rank</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim</td>
<td>C4</td>
<td>43</td>
</tr>
<tr>
<td>John</td>
<td>C3</td>
<td>33</td>
</tr>
<tr>
<td>Ken</td>
<td>C4</td>
<td>57</td>
</tr>
<tr>
<td>Lisa</td>
<td>C4</td>
<td>39</td>
</tr>
<tr>
<td>Tom</td>
<td>C2</td>
<td>32</td>
</tr>
<tr>
<td>Eva</td>
<td>C3</td>
<td>36</td>
</tr>
</tbody>
</table>

GROUP BY $P.Rank$

<table>
<thead>
<tr>
<th>Name</th>
<th>Rank</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim</td>
<td>C4</td>
<td>43</td>
</tr>
<tr>
<td>Lisa</td>
<td>C4</td>
<td>39</td>
</tr>
<tr>
<td>John</td>
<td>C3</td>
<td>33</td>
</tr>
<tr>
<td>Eva</td>
<td>C3</td>
<td>36</td>
</tr>
<tr>
<td>Tom</td>
<td>C2</td>
<td>32</td>
</tr>
</tbody>
</table>

GROUP BY $P.Rank$

<table>
<thead>
<tr>
<th>Rank</th>
<th>AvgAge</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>32.0</td>
</tr>
<tr>
<td>C3</td>
<td>34.5</td>
</tr>
<tr>
<td>C4</td>
<td>41.0</td>
</tr>
</tbody>
</table>
• **We need aggregate functions in Datalog** as well – they ought to be treated similarly with other functions (at least as far as they exhibit comparable properties), e.g., terms containing aggregate functions should appear in test literals only.

• **But aggregates are considerably different** from, e.g., arithmetic functions as they require the prior computation of „the aggregate“, i.e., the collection of objects to which they are to be applied.

• Therefore, we need a special construct expressing an aggregate inside the body of the rule, which means a kind of nesting „hidden“ by a special syntax, e.g.:

\[
\text{avg\_salary}(\text{Dept}, \text{AvgS}) \leftarrow \text{AvgS} = \text{avg}(\text{Salary}, \text{Dept}, \text{employee}(E, \text{Dept}, \text{Salary}))
\]

• There are special ternary aggregate functions: avg, sum, max, min, card.  
(We prefer \textit{cardinality} rather than \textit{count}.)

• The 1\textsuperscript{st} parameter of each aggregate term is the variable to be aggregated about.
• The 2\textsuperscript{nd} parameter is the \textit{grouping} variable.
• The 3\textsuperscript{rd} parameter is a literal defining the grouping condition.
• Thus, the example reads: AvgS is the average salary per department computed from the \textit{employee} relation.
Summary of Notations and Concepts

fact
rule
    rule head
    rule body
literal
    ground literal
    retrieval literal
    test literal
safety
conjunction, disjunction
negation
quantifier
variable
    anonymous variable
    local variable
order independence
instance
    ground instance

CWA
negation as failure
dependency graph
recursion
standardization
built-ins
aggregates