Intelligent Information Systems

PD Dr. Andreas Behrend
University of Bonn

Acknowledgements
I am indebted to Prof. Dr. Rainer Manthey for providing me slides for these lecture notes.
Who for Whom

**CLASSES AND EXERCISES**

- For Master students: Informatik, Bioinformatik

- Class 2 SWS: Every Tuesday, 16:15 to 17:45 (HS 3.31)
- Exercise 2 SWS starting April 13th: Wednesdays, 8:30 to 10:00 (SR 1.03)
- Credits: 2+2

**LECTURE NOTES**

- Available at [http://www.iai.uni-bonn.de/III/lehre/vorlesungen/FIM/SS16/IIS](http://www.iai.uni-bonn.de/III/lehre/vorlesungen/FIM/SS16/IIS)
- News only via homepage of the lecture
Basic Rules

**EXERCISES**
- You are strongly encouraged to attend the exercise sessions but it is not mandatory to do so.
- The solutions to the exercises will *not* be published.

**FINAL EXAM**
- The oral examination will be strongly influenced by the tasks discussed during the exercises.
- You have to ask if you are required to have a written examination.
1. Datalog - Syntax
Datalog

Facts

\[
\begin{align*}
\text{p(1,a).} \\
\text{p(2,b).} \\
\text{p(3,c).} \\
\text{q(2).} \\
\text{q(5).} \\
\text{r(a,1).} \\
\text{r(a,2).} \\
\text{r(b,3).}
\end{align*}
\]

Constants

\[
\begin{align*}
\text{s(X)} & \leftarrow \text{p(X,Y)}.
\text{s(X)} & \leftarrow \text{r(Y,X)}.
\text{t(X,Y,Z)} & \leftarrow \text{p(X,Y)}, \text{r(Y,Z)}.
\text{w(X)} & \leftarrow \text{s(X)}, \text{not q(X)}.
\end{align*}
\]

Rules

Relation Names

\[
\begin{align*}
p, \ q, \ r: \ Base \ relations
\end{align*}
\]

\[
\begin{align*}
s, \ t, \ w: \ Derived \ relations
\end{align*}
\]
Datalog: Applying Rules to Facts (1)

s(X) ← p(X,Y).
s(X) ← r(Y,X).
t(X,Y,Z) ← p(X,Y), r(Y,Z).
...

s, t: Derived relations

Rules

Facts

p(1,a).
p(2,b).
p(3,c).
...
r(a,1).
r(a,2).
r(b,3).
Datalog: Applying Rules to Facts (2)

Facts

\[ p(1,a). \]
\[ p(2,b). \]
\[ p(3,c). \]
\[ \ldots \]
\[ r(a,1). \]
\[ r(a,2). \]
\[ r(b,3). \]

Rules

\[ s(1) \leftarrow p(1,a). \]
\[ s(1) \leftarrow r(a,1). \]
\[ t(1,a,1) \leftarrow p(1,a), r(a,1). \]
\[ \ldots \]

s, t: Derived relations
Datalog: Applying Rules to Facts (3)

Facts

s(1)
s(2)
t(1,a,1)
t(1,a,2)

Rules

s(2) ← p(2,b).
s(2) ← r(a,2).

... t(1,a,2) ← p(1,a), r(a,2).

p(1,a).
p(2,b).
p(3,c).
...

r(a,1).
r(a,2).
r(b,3).

s, t: Derived relations
Datalog: Applying Rules to Facts (4)

Rules

\[
\begin{align*}
  s(3) & \leftarrow p(3, c). \\
  s(3) & \leftarrow r(b, 3). \\
  t(2, b, 3) & \leftarrow p(2, b), r(b, 3). \\
  \ldots
\end{align*}
\]

Facts

\[
\begin{align*}
  p(1, a). \\
  p(2, b). \\
  p(3, c). \\
  \ldots \\
  r(a, 1). \\
  r(a, 2). \\
  r(b, 3). \\
\end{align*}
\]

s, t: Derived relations
Deductive Rules

- Derivable information in a database is only possible, if the DBMS supports a definition language for derived data/views.

- View definitions are part of the DB schema; they are expressed using the DDL of the respective DBMS (e.g., the CREATE VIEW command in SQL).

- "View definition" and "Deductive rule" are synonyms; in this lecture, we will speak about deductive rules most of the time.

- General structure of each deductive rule:

  
<table>
<thead>
<tr>
<th>Rule head</th>
<th>Rule body</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern of an information unit of a derivable set of data</td>
<td>defining query</td>
</tr>
</tbody>
</table>

- Deductive rules are declarative expressions (like queries).
Data Model and Rule Language

- In principle, each data model can be extended by deductive and normative rules (on the basis of the resp. query language).

- In this lecture, we are going to use the relational model exclusively. But rules for other models like the ER model, XML and RDF data, or object-oriented models have been proposed, too.

- View definitions may look very different, depending on the query language they are based on (even for the same data model), e.g.:

  CREATE VIEW p
  AS ( SELECT q.A
       FROM   q, r
       WHERE  q.B = r.C )

  p(X) ← q(X,Y), r(Y,Z).

  \[ P = \pi_A (Q \bowtie R) \text{ (} B=C \text{)} \]

Different syntax, but identical semantics!
Advantages of using deductive rules:

- Self-documenting (declarative ⇒ "readable")
- Easily changeable ⇒ flexible
- Modular knowledge representation, incrementally constructable
- Simple, efficient representation of useful terminology without redundant data storage
- "Business rules" of the application are made explicit.

Additional advantages of virtual data representation:

- Sometimes enormous savings in space
  (often „paid“ by additional costs for time during query processing)
- Immediate "visibility" of implicit changes
  (caused by explicit changes of base data)
Datalog Language Summary

- "Datalog" : "Database + Prolog", notion coined by 1984

- **Syntactically**: Strong influence by logic programming language Prolog (but: Only simple form of "pure" Prolog adapted)

- **Semantically**: Strongly set-oriented like other languages, e.g., SQL or RA (instead of instance-oriented like Prolog)

- **„Lingua franca“ in research** on deductive databases („de facto“ standard)
  - But: Up till now **not** used commercially, **no** standardisation

- Rather **uniform** syntax and semantics of facts and rules
- Various **different** proposals for queries, updates, constraints and schemas

- Datalog is based on the relational **domain calculus (DRC)**, but uses just a **minimal** set of logical operators:
  - **Conjunction** and **Negation**
Facts in Datalog

Facts are represented by atomic formulas, the parameters of which are all constants.

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
<th>Car</th>
<th>Inhabitants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berlin</td>
<td>030</td>
<td>B</td>
<td>3399</td>
</tr>
<tr>
<td>Hamburg</td>
<td>040</td>
<td>HH</td>
<td>1700</td>
</tr>
<tr>
<td>München</td>
<td>089</td>
<td>M</td>
<td>1189</td>
</tr>
<tr>
<td>Köln</td>
<td>0221</td>
<td>K</td>
<td>963</td>
</tr>
<tr>
<td>Frankfurt</td>
<td>069</td>
<td>F</td>
<td>644</td>
</tr>
<tr>
<td>Essen</td>
<td>0201</td>
<td>E</td>
<td>603</td>
</tr>
</tbody>
</table>
Constants, Variables, Relation Names

- **Variables:** Capital letters or strings beginning with a capital:
  
  e.g.: \(X\) \(X_1\) City \(X1a2b\%\)

- **Constants:** Digits, lower case letters, or strings beginning with a lower case letter or a digit, . . .
  
  e.g.: \(x\) 1 city 1a2b\%

  . . . or arbitrary strings in apostrophes:
  
  e.g.: 'City' 'X' '?-abc-!'

- **Relation names:** Strings beginning with a lower case letter
  
  e.g.: city p q_1
Deductive Rules

Rule head
population(City, Inhabitants)
←
city(City, Phone, Car, Inhabitants).

Rule body

Derived relation
population('Berlin',3399).
population('Hamburg',1700).
population('München',1189).
population('Köln',963).
population('Frankfurt',644).
population('Essen',603).

Base relation
city('Berlin',030,'B',3399).
city('Hamburg',040,'HH',1700).
city('München',089,'M',1189).
city('Köln',0221,'K',963).
city('Frankfurt',069,'F',644).
city('Essen',0201,'E',603).
The basic constituents of all Datalog-expressions are positive or negative atomic formulas, from which facts and rules are built.

Such formulas are called literals.

- **Positive literals**
  - `p(X,Y)`
  - `q(a, Y, 1)`
  - `r(a,b)`

- **Negative literals**
  - `not p(X,Y)`
  - `not q(a,Y,1)`
  - `not r(a,b)`

Rule heads and facts are positive literals; rule bodies may contain both, positive and (possibly) negative literals.

Literals without any variables are called ground literals. Therefore, all facts are ground literals in Datalog.
Rule Application: Principle

Transfer of variable bindings from body to head

Rule head serves as a pattern for derivable facts

Evaluation of the rule body produces variable bindings

Derived facts

Population(C,I) ← city(C,P,Car,I).

Population('Berlin',3399).

• • •

Base facts or previously derived facts

City('Berlin',030,'B',3399).

• • •
Safety of Rules

• Rules are able to serve as "fact producers" only if the evaluation of the rule body generates variable bindings for all variables in the rule head.

• Thus: All variables in the head have to appear in the body of a rule, too!

• Rules satisfying this requirement are called safe.

All variables in the rule head . . .

\[
\begin{align*}
\text{population} & \leftarrow \text{city} \\
\text{city} & \leftarrow \text{city}
\end{align*}
\]

. . . have to appear in the rule body, too!

(But not necessarily vice versa!)
Unsafe Rules

- An unsafe rule (like this one) is not able to produce complete facts for the defined relation 'in_state':

  Where to take values for binding variable 'State' from?

- In principle: For 'State' any constant value could be substituted, so that the resulting relation 'in_state' ought to be infinitely large!

- In this lecture: All rules are assumed to be safe – no unsafe rules!
Conjunctive Rules (Joins)

**is_a_capital('München', 'M').**

Comma as $\land$-operator

Concatenation via identical variables

Substitution with identical values

**capital_of('Bavaria', 'München').**

**city('München', 089, 'M', 1189 ).**
Relation Defined by Several Rules

Derived Relations may be defined by **more than one rule**:

\[
\text{European city}(X) \leftarrow \text{situated in}(X, 'Denmark').
\]

\[
\text{European city}(X) \leftarrow \text{situated in}(X, 'Germany').
\]

\[
\text{European city}(X) \leftarrow \text{situated in}(X, 'Russia').
\]

.....

Each rule is able to derive a **partial relation**.

Some derived facts may be simultaneously derivable by **several** rules. As relations are sets, just one „copy survives“!

The entire relation thus is the **union** of all these partial relations (i.e., subsets of the full relation).

"Union semantics"
Order Independance in Datalog

• Datalog has been conceived as a purely declarative language, for which any aspect of execution – in particular of efficient evaluation – is irrelevant for the definition of syntax and semantics.

• Thus: The order of notation is irrelevant . . .
  • . . . if several rules define the same relation.
  • . . . if several literals occur in a rule body.

• The following rule sets are considered equivalent in Datalog, even though they are syntactically different:

\[
\begin{align*}
p(X) &\leftarrow q(X,Y), \text{not } s(Y). \\
p(X) &\leftarrow r(X), t(X), w(X).
\end{align*}
\]

\[
\begin{align*}
p(X) &\leftarrow t(X), w(X), r(X). \\
p(X) &\leftarrow \text{not } s(Y), q(X,Y).
\end{align*}
\]

• For any concrete evaluation strategy, however, an evaluation order for literals and rules has to be fixed, though, and to be planned well for efficiency’s sake.
Anonymous Variables (1)

- Frequent situation in Datalog-rules:
  Various local variables are not used for connecting literals or for representing output values, but just for „filling“ parameter positions.

```
is_a_capital(City, Car) ←
capital_of(State, City),
city(City, Phone, Car, Inhabitants).
```

- Adopted from Logic Programming:
  Abbreviating notation for this kind of "fill-up parameters" by underscore.

```
is_a_capital(City, Car) ←
capital_of(_, City),
city(City, _, Car, _).
```

"anonymous variables" (aka: "don't care"-variables)
Anonymous Variables (2)

• Although the same symbol (underscore) is used for representing anonymous variables, each occurrence of such a variable stands for a different, completely new variable occurring in this position only:

\[
\text{is\_a\_capital}(\text{City, Car}) \leftarrow \\
\text{capital\_of} (\_ , \text{City}) , \text{city}(\text{City, }\_ , \text{Car, }\_).
\]

\[
\exists X_1, X_2, X_3
\]

• Each of these variables has its own implicit quantifier. All these quantifiers is placed at the beginning of the rule body.
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 the 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).
CWA

**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

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, \textit{not} in facts and \textit{not} in the head of a rule (i.e., not for derivable facts).

• There is \textit{no} directly expressible negative information in a Datalog-DB:

  \textbf{not} capital\_of('Germany', 'Bonn') .

  No stored "negative facts" !!

• \textbf{Derivation} of negative information is excluded, too:

  \textbf{not} is\_a\_capital( X ) \leftarrow situated\_in( X, 'Bavaria' ) .

  No derivable "negative facts" !!
Negation In Rules

• 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):

  ```
  north_german_city(X) ← 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: **and not**

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

  ```
  north_german_city(X) ←
  city(X, Y1, Y2, Y3), not is_situated_in(X, 'Bavaria').
  ```

  Production  Test
Evaluating Negative Literals

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

\[
\text{normal\_city}(\text{City}, \text{Car}) \leftarrow \text{city}(\text{City}, \_ , \text{Car}, \_ ) , \text{not} \text{ is\_a\_capital}(\text{City}, \text{Car}).
\]

\[
\text{city}(\text{'Köln'}, 0221, \text{'K'}, 963).
\]

\[
\text{city}(\text{'München'}, 089, \text{'M'}, 1189).
\]

\[
\text{is\_a\_capital}(\text{'Köln'}, \text{'K'}) ?
\]

\[
\text{is\_a\_capital}(\text{'München'}, \text{'M'}).
\]

\[
\text{is\_a\_capital}(\text{'Düsseldorf'}, \text{'D'}).
\]

Positive „side evaluation“
Negation as Failure

• 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 not F, . . .
  - . . . try to answer ist positive part F.
  - If F is true, then not F is false.
  - If F is false, then 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)'
  - 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 in a separate Regel:

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

Auxiliary relation
No Nesting in Datalog

• 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').
Hierarchical Dependencies

Important in particular for constructing terminological hierarchies:

Derived relationen may depend on other derived relations (not only on base relations), i.e.:
In the body of a rule, arbitrary relations may be referenced by literals.

e.g.:

\[
\begin{align*}
p(X,Y) & \leftarrow q(X,Y), r(Y,X). \\
q(X,Y) & \leftarrow s(X,Y,Z). \\
r(Y,X) & \leftarrow t(X,Y), \text{not } w(Y). 
\end{align*}
\]

Corresponding dependency graph:
Recursive Dependencies

- **Moreover**: Rule-defined relations may depend on themselves, i.e., rules may be **recursive**.

- **But**: Recursive rules ought to come with at least one non-recursive rule defining the same relation ("well-founded recursion")

### Example

- **Non-recursive**
  
  \[
  \text{reachable_from}(A, B) \leftarrow \text{adjacent}(A, B).
  \]

- **Recursive**
  
  \[
  \text{reachable_from}(A, B) \leftarrow \text{adjacent}(A, C),
  \text{reachable_from}(C, B).
  \]

*Restriction to be lifted later on!*
In SQL, "empty fields" in a table are permitted, in case information about the particular attribute is missing for the particular object represented in the resp. row.

Theoretically, empty fields are considered to contain a special value – called NULL, or: a null value – representing an "existing, but unknown" value in the resp. domain.

Thus, NULL cannot (or better: ought not) be used for representing cases where, e.g., the resp. attribute does not apply, or where we do no know whether a value exists at all for the resp. field.

Null values introduce quite sophisticated problems for query evaluation to SQL – ultimately, a 3-valued logic (with an extra value "unknown") has to be introduced in order to properly deal with the implications of using NULL in the particular interpretation fixed in the SQL standard.

Even though nulls are rather helpful in many practical cases, we do not allow NULL in Datalog, due to the problems resulting. Researchers in deductive DBs have been agreeing on this till now.

No NULL in Datalog!
Standardisation

- A **deductive database** in Datalog is a set of facts and rules (later on in this lecture, we will reconsider this "definition" a bit).

- It is usually assumed 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:

```
p(a).
p(b).
p(X) ← q(X,Y).
p(X) ← s(X), t(X).
```

```
p1(a).
p1(b).
p1(X).
p(X) ← p1(X).
p(X) ← q(X,Y).
p(X) ← s(X), t(X).
```

Not admissible!

Standardized
Built in-Predicates

• 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.

  \[
  x > y \quad x \leq 1 \quad \text{not} \ a = b
  \]
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**
    - \( p(X,Z) \land X > Y \land q(Z,Y) \)
  
  - **Unsafe**
    - \( p(X,Y) \land Y < Z \)
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:

  ⇒ **Functional terms are (for now) admissible in test literals only!**

- **Reason:** Test literals are to be evaluated externally anyway; moreover, functions rely on "safety" of all parameters, too.

  ![Not really tests anymore!](image)
Aggregate Functions

• **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.:

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

- There are special ternary aggregate functions: avg, sum, max, min, card.  
  (We prefer \text{card(inality)} rather than \text{count}.)
- The 1\text{st} parameter of each aggregate term is the variable to be aggregated about.
- The 2\text{nd} parameter is the grouping variable.
- The 3\text{rd} parameter is a literal defining the grouping condition.
- Thus, the example reads: AvgS is the average salary per department computed from the employee relation.