Learning by Doing (cont.)

Intelligent Information Systems

WS 2016/17

2. Datalog and SQL

2.1 Datalog and SQL: Learning by Doing
2.2 Datalog: Syntax and (Basic) Semantics
Using Negation in Datalog and SQL (Reminder)

• In our initial example comparing Datalog and SQL (when declaring the same derived relations/tables) we used one rule/view containing logical negation:

\[
\ldots \\
w(X) \leftarrow s(X), \textbf{not} q(X).
\]

\[
\ldots \\
\text{CREATE VIEW } w \text{ AS } \\
\text{(TABLE } s) \\
\text{MINUS } \\
\text{(TABLE } q);
\]

• In Datalog, the Boolean operator \textbf{not} appears in the rule body next to the operator and written in Datalog-style as a comma symbol. In the SQL view declaration the set-theoretic counterpart \textbf{MINUS} is used instead.

• \textbf{Negation} (regardless in which syntactic variant) potentially causes \textbf{trouble} in a deductive (relational) database – to be explained later. Therefore we will have to treat negation \textbf{with particular care}!

• SQL knows logical NOT, too. How to express this example rule in SQL using NOT?
In (relational) databases:

+ All stored facts are (supposed to be) true.
+ All true facts are (supposed to be) stored (or derivable).

- All non-stored or non-derivable facts are (supposed to be) false.
- No false facts are (supposed to be) stored (or derivable).
• We need a bit of preparation before being able to use negation in our genealogy context.

• In the exercises, we will try to formalize the concept of being (currently) married – a rather difficult affair. Now let us define who was ever married, and who has at least one child. For making things easier, we just look at the male case (husband, 1\textsuperscript{st} parameter of marriage as well as child):

  \[
  \text{ever\_married}(P) \leftarrow \text{marriage}(P,\_,\_).
  \]

  \[
  \text{has\_child}(P) \leftarrow \text{child}(P,\_,\_).
  \]

• Now for the rule requiring negation: An uncle who has no children and was never married is interesting (because we might inherit his fortune, when he dies!).

  \[
  \text{interesting\_uncle}(N,U) \leftarrow \text{uncle}(N,U),
  \quad \text{not} \quad \text{ever\_married}(U),
  \quad \text{not} \quad \text{has\_child}(U).
  \]
How to compute interesting uncles if you only have positive data at hand?
Although we do not have data about persons who never married, or are childless . . .

\[
\text{interesting_uncle}(N, U) \leftarrow \\
\text{uncle}(N, U), \\
\text{not} \text{ ever_married}(U), \\
\text{not} \text{ has_child}(U).
\]

. . . we can use the positive data about persons who did marry or do have a child for eliminating uncles who definitely are not interesting! No need to „access“ the complement of any table explicitly.
interesting_uncle(N,U) ← uncle(N,U), not ever_married(U), not has_child(U).

At the end, only Henry is interesting (for George)!
In SQL, the corresponding query defining the view requires two embedded subqueries correlated with the main query by means of \texttt{NOT IN} (or, similarly, \texttt{NOT EXISTS}):

\begin{verbatim}
CREATE VIEW Interesting_uncle AS
    (SELECT * FROM Uncle
     WHERE Uncle NOT IN
         (SELECT * FROM Ever_married)
     AND Uncle NOT IN
         (SELECT * FROM Has_child))
\end{verbatim}

The \textit{evaluation strategy} is the same in SQL: Reduce \textit{Uncle} by \textit{eliminating} those rows for which the \texttt{NOT IN} condition does \textit{not} hold!
Open Question Remaining about Negation

Why not simply use one rule rather than three?

There is a good – even though debatable – reason for using the three rules rather than one.

This will be discussed in the next section, please wait!
Traversing Family Trees: Fixed Number of Steps (1)

Don’t be surprised, this is the Royals family tree image from last year.
Traversing Family Trees: Fixed Number of Steps (2)

1 step down the tree
(in direction of ancestors, i.e. towards the root of the family tree)

2 steps down the tree

3 steps down the tree

What to do if we don't even know how deep we can go in the tree?

parent(N,P) ← child(P,_,N).
parent(N,P) ← child(_,P,N).

grandparent(N,GP) ← parent(N,P), parent(P,GP).

great-grandparent(N,GP) ← parent(N,P), grandparent(P,GP).

These arrows are not referring to the family tree, but represent dependency of relations on each other!
Traversal of a family tree down to arbitrary depth can be expressed very elegantly using the most powerful syntactic feature of declarative query languages: **Recursion**

**Datalog:**

\[
\begin{align*}
\text{ancestor}(X,Y) & \leftarrow \text{parent}(X,Y). \\
\text{ancestor}(X,Y) & \leftarrow \text{parent}(X,Z), \text{ancestor}(Z,Y).
\end{align*}
\]

**SQL:**

```
CREATE RECURSIVE VIEW Ancestor AS
(SELECT Name, Parent AS Ancestor
FROM Parent)
UNION
(SELECT P.Name, A.Ancestor
FROM Parent AS P,
Ancestor AS A
WHERE A.Name = P.Parent))
```

Recursive views are allowed in SQL since the standard of 1999!
All the children of a person X are ancestors of X, as well as all ancestors of the children of X (and so on):

*Ancestor* is a recursively defined concept!

\[
\text{ancestor}(X,Y) \leftarrow \text{parent}(X,Y).
\]
\[
\text{ancestor}(X,Y) \leftarrow \text{parent}(X,Z), \text{ancestor}(Z,Y).
\]

*Descendant* is the inverse concept of ancestor:

\[
\text{descendant}(X,Y) \leftarrow \text{ancestor}(Y,X).
\]

The generation (or level) of an ancestor can be expressed by means of an additional parameter, which is recursively incremented:

\[
\text{ancestor}(X,Y, 1) \leftarrow \text{parent}(X,Y).
\]
\[
\text{ancestor}(X,Y, J) \leftarrow \text{parent}(X,Z), \text{ancestor}(Z,Y, I), \ J = I+1.
\]
\[
\text{descendant}(X,Y, I) \leftarrow \text{ancestor}(Y,X, I).
\]
Computation of the ancestor table is done iteratively.

At the beginning, there are no ancestor facts yet, so that the recursive rule cannot “produce“ anything.

Just the non-recursive rule is able to provide an initial bunch of ancestor facts „copied“ from parent.

```
ancestor(X,Y,1) ← parent(X,Y).
ancestor(X,Y,J) ← parent(X,Z), ancestor(Z,Y,I), J = I+1.
```
Computing Royal Ancestors Recursively (2)

\[
\text{ancestor}(X,Y,1) \leftarrow \text{parent}(X,Y).
\]
\[
\text{ancestor}(X,Y,J) \leftarrow \text{parent}(X,Z), \text{ancestor}(Z,Y,I), J = I+1.
\]

First application of the recursive rule produces further ancestor facts.
Computing Royal Ancestors Recursively (3)

```
ancestor(X,Y,1) ← parent(X,Y).
ancestor(X,Y,J) ← parent(X,Z), ancestor(Z,Y,I), J = I+1.
```

Similarly in iteration 3: Combining generation 2 facts with parent facts.
Nothing new in iteration 4: stop!
Computing Royal Ancestors Recursively (4)

\[
\text{ancestor}(X, Y) \leftarrow \text{parent}(X, Y).
\]

\[
\text{ancestor}(X, Y) \leftarrow \text{parent}(X, Z), \text{ancestor}(Z, Y).
\]

30 facts + 22 facts + 6 facts = 58 facts
The degree of kinship (being relative or relative-in-law) is determined by the number of intermittent births.
The concept "relative of" can be defined via several Datalog rules, too, using the concepts ancestor and descendant introduced before:

\begin{align*}
\text{relative}(X, Y, \text{Degree}) & \leftarrow \\
& \text{ancestor}(X, Y, \text{Degree}). \\
\text{relative}(X, Y, \text{Degree}) & \leftarrow \\
& \text{descendant}(X, Y, \text{Degree}). \\
\text{relative}(X, Y, \text{Degree}) & \leftarrow \\
& \text{ancestor}(Z, X, \text{Degree1}), \\
& \text{ancestor}(Z, Y, \text{Degree2}), \\
& \text{not ancestor}(X, Y), \\
& \text{not ancestor}(Y, X), \\
& \text{not has_younger_common_anc}(X, Y, Z), \\
& X \neq Y, \\
& \text{Degree} = \text{Degree1} + \text{Degree2}. \\
\text{has_younger_common_anc}(X, Y, Z) & \leftarrow \\
& \text{ancestor}(Z_1, X, _), \\
& \text{ancestor}(Z_1, X, _), \\
& \text{ancestor}(Z, Z_1, _).
\end{align*}