Intelligent Information Systems

WS 2018/19

Deduction in Datalog and SQL

Chapter 2
• The essential topic of this lecture is to understand the importance of the idea of using the concept of derived data (and of the corresponding technology).

• We already learnt last week, that databases „containing“ stored as well as derived data are called deductive databases in scientific terminology.

• In order to manage derived data, we need special abilities of our DBMS that turn it into a deductive DBMS.

• Managing derived data means
  • designing a deductive DB schema
  • incl. specifying derivation rules
  • evaluating queries over derived data
  • controlling consequences of changes of derived data
• **Storing** data is the „normal“ way of keeping data (in some storage device).

• Every data element **can be stored** (in principle).

• But there are data elements that do **not** (necessarily) **have to be stored**, but (possibly) can be computed from stored (or other) derived data elements instead. We call such pieces of data „**derived**“ data (elements).

• It would be more precise to speak about „**derivable**“ data, because may be a derivable piece of data is actually **never derived** later on (or has never been derived up till now). We will use the term „**derived**“ data nevertheless.

• Part of the derivable data can redundantly be stored (after having been derived first) in order to avoid costly rederivation. We call such data **materialized** (derived) data.
How do *inference* (derivation) *engines* look like? Should they be *part of a DBMS* or external tools?

We follow a *specific* approach in IIS!
Intelligent Information Systems

WS 2018/19

2. Datalog and SQL

2.1 Datalog and SQL: Learning by Doing
2.2 Datalog: Syntax and (Basic) Semantics
2.3 From SQL to Datalog (and Back)
A case study on using derived data will give us a concrete start into using the techniques of Deductive Database technology: A Genealogical Database
The Royal Family Tree: Our Example Application Domain

The current family tree of the British royal family (starting from the Queen and her husband)

Parent-Child

Husband-Wife
Since last year, several "updates" happened:
- William and Kate have a third child, Louis
- Harry married Meghan
- Eugenie married Jack
- (to come: Meghan is pregnant ...)
From Family Tree to Genealogical Database?

How to „put a family tree into a (relational) database“?

(i.e., how to design a basic genealogical database)
• Traditional way of designing a relational DB:
  Start by **conceptually modelling** the resp. application domain

• Corresponding **ER diagram** (Entity-Relationship approach):

  ![Family Tree ER Diagram]

  - Basic technique for **logical design**: deriving a **relational DB schema** from an ER diagram
    - per **entity type** one table: Each attribute of the E-type turned into attribute of table
    - per **relationship type** on table, too: Each attribute of the R-type into attribute of corr. table, plus key attributes of of participating E-types (as foreign keys)
The basic relational schema for this ER diagram looks as follows:

- person (title, name, birth, death, sex)
- parents (father, mother, child)
- marriage (husband, wife, married, divorced)

Roles in R-types are turned into attributes of the corr. R-table.
• person (title, name, birth, death, sex)
• parents (father, mother, child)
• marriage (husband, wife, married, divorced)

The initial design can be improved by integrating certain relationship tables into entity tables in case of 1:N functionalities:

- person (title, name, birth, death, sex, father, mother)
- marriage (husband, wife, married, divorced)
One possible (initial) relational format for family trees: just two tables!

The entire family tree is „in“ these two tables.

Disadvantage(?): Many empty cells („nulls“)
Just Stored Data in Our Genealogical DB – Up Till Now

- All the data representing what we know about the Royal family members (in the family tree) have to be stored in tables of our relational DB.
- All the information in these two tables are „given“ data reported from the „real world“.
- None of this could have been derived from other parts of the database.

Can we turn this DB into a deductive DB by extending it with additional derived data?

How to do this, ... if we can?
• Data about „existence“ (birth, death, sex, name etc.) cannot be derived, but have to be entered manually and stored in tables. The same applies to marriage and divorce.
• All other data about family relationships, however, are derivable from these stored data, provided we can express them using suitable derivation rules.
Example: Deriving Uncle Data from the Stored Family Tree Data

How to derive data about the uncles of a person?

(Aunts could be treated analogously.)

Uncle (from Latin: *avunculus* "little grandfather", the diminutive of *avus* "grandfather") is a family relationship or kinship, between a person and his or her parent's brother, parent's brother-in-law or parent's cousin. (from: en/wikipedia)

(Here we restrict ourselves to uncle in the narrow sense)
It would be nice to have the data about uncles in the Royal family in a table of its own – best in a derived table! (If possible!)

(Uncles of Charlotte and Mia still missing.)
An uncle of a person is (one of) his (or her) parent's brothers.
Uncle Relationships in the Royal Family Tree!

And several additional uncle relationships – how many?

(Do Savannah and Isla have any uncles?)
How to Derive a Particular Uncle Row?

How to derive the Uncle table from the Person table?

A derivation strategy a human would use, might serve as a suitable strategy of a deduction "engine".

(Is this some kind of "artificial intelligence"?)
Relational Views as Specifications of a Derivation Strategy

Derived tables in a relational DB are obtainable as the result of a query. Each query thus expresses a deduction „strategy“ of a table.

Named queries can be stored in a relational DB, introducing a named derived table. In SQL, these queries are called views.

How does the SQL view look like that can „do“ this kind of deduction?
CREATE VIEW Uncle AS

(SELECT P1.Name,
     P3.Name AS Uncle
FROM Person AS P1,
     Person AS P2,
     Person AS P3
WHERE P1.Father = P2.Name
     AND P2.Mother = P3.Mother
     AND P2.Father = P3.Father
     AND P2.Name <> P3.Name
     AND P3.Sex = 'm')

UNION

(SELECT P1.Name,
     P3.Name AS Uncle
FROM Person AS P1,
     Person AS P2,
     Person AS P3
WHERE P1.Mother = P2.Name
     AND P2.Mother = P3.Mother
     AND P2.Father = P3.Father
     AND P2.Name <> P3.Name
     AND P3.Sex = 'm')

Apply this code to the Person table on the previous slide!
"Uncle" in SQL – 2\textsuperscript{nd} Attempt

The direct derivation of \textit{Uncle} from the base table \textit{Person} is very complex – why not try to simulate the Wikipedia definition, which uses auxiliary concepts „parent“ and „brother“?

\textbf{Uncle} ... is a family relationship ... between \textbf{a person and his or her parent's brother} ...

Assume we had views \textit{Parent(Name, Parent)} and \textit{Brother(Name,Brother)} available – then \textit{Uncle} could be specified like this:

\begin{verbatim}
CREATE VIEW Uncle AS
(SELECT P.Name AS Name,
B.Brother AS Uncle
FROM Parent AS P,
Brother AS B
WHERE P.Parent = B.Name)
\end{verbatim}

Much, much \textbf{simpler} – but we still need specifications of the two other views ...

e.g.:

\begin{align*}
\text{Parent}(\text{'}Peter', \text{'}Anne') & \quad \{ \quad \text{Uncle}(\text{'}Peter', \text{'}Charles') \\
\text{Brother}(\text{'}Anne', \text{'}Charles')
\end{align*}
• Both, father and mother of a person are the *parents* of this person.

• In English, the term *parent* exists as a singular noun, too – unlike, e.g., in German.

• So the SQL specification of *Parent* needs two cases, one for paternal and maternal parent each:

```
CREATE VIEW Parent AS
    (SELECT Name, Father AS Parent
        FROM Person)
UNION
(SELECT Name, Mother AS Parent
    FROM Person)
```

*paternal parent*

*maternal parent*
• The *brother* of a person is a male with exactly the same pair of parents.

• If just one of mother or father is the same, we speak of a *half-brother*.

• The SQL view *Brother* looks like this:

```
CREATE VIEW Brother AS
    (SELECT    P1.Name,
             P2.Name  AS Brother
    FROM Person AS P1,
         Person AS P2,
    WHERE P1.Mother = P2.Mother
         AND     P1.Father = P2.Father
         AND     P1.Name <> P2.Name
         AND     P2.Sex = 'm')
```
Two-level Specification of „Uncle“

CREATE VIEW Uncle AS
(SELECT P.Name AS Name,
 B.Brother AS Uncle
FROM Parent AS P,
 Brother AS B
WHERE P.Parent = B.Name)

CREATE VIEW Parent AS
((SELECT Name,
Father AS Parent
FROM Person)
UNION
(SELECT Name,
Mother AS Parent
FROM Person))

CREATE VIEW Brother AS
(SELECT P1.Name,
P2.Name AS Brother
FROM Person AS P1,
 Person AS P2,
WHERE P1.Mother = P2.Mother
AND P1.Father = P2.Father
AND P1.Name <> P2.Name
AND P2.Sex = 'm')

Create View based on two „lower level“ views.

Views based on two tables.
Using views enables us to extend a given database of stored tables by means of additional derived tables that are automatically generated by the DBMS on demand.

We call this generation process deduction (as it follows laws of inference invented in logic). Thus, databases using views are deductive databases.

View definitions are pieces of code in SQL – i.e., you „program“ SQL, if specifying views. View definitions are the counterpart to procedures (or methods) in imperative programming.

If using views properly, each view is a declarative specification of a concept (or: a term) of the respective application domain. Each of these specifications has to be constructive, i.e., usable for generating all instances of the concept defined over the given base data.

It is not (always) easy to correctly and completely specify a concept! You need a precise understanding of the definition of the resp. concept in natural language before coding SQL.

Quite often there are various alternatives how to formulate a specification. They may differ in elegance of style, ease of understandibility and – most important – in efficiency of organizing deduction processes.

A multi-level specification, using intermediate concepts, is often preferable.
Datalog vs. SQL: Some Basics Ahead

- Research in deductive databases has a nearly 40-years history (as old as SQL), but has been using a different declarative language (not SQL!) most of the time, strongly influenced by the logic programming language PROLOG:

  **Datalog**

- Nearly all publications in this area have been using Datalog – that’s why we will use Datalog during this lecture, too (and you will have to learn it!).

- Many results of DDB research have been transferred to the SQL world recently! That’s why SQL will also be appearing throughout the lecture in various places.

**SQL:**

- used in industry and commerce
- supported by many DBMS products
- standardized
- user-friendly („controlled English“)
- rich set of syntactic features

**Datalog:**

- used in academia only
- just few academic prototypes
- no standards
- mathematical style
- minimalistic syntax
• In 2.1: Informal introduction to Datalog by means of an extended example.

• Simultaneously: Example-based introduction to . . .
  • . . . specifying concepts in a declarative DB language
  • . . . answering queries over deductive rules
  • . . . deduction using SQL views

• At first:
  • No rigorous treatment of concepts and ideas – „wetting“ your appetite is the goal.
  • Just core concepts presented and used in examples, more to come.

• In 2.2: More in-depth systematic treatment of Datalog (in full).

• Aim: Enable students to start „speaking“ the language straightaway.

• So, do make use of this chance: Exercise yourselves – don‘t wait to be „forced“!

• There is no use reflecting about theory if you do not have any first-hand experience in practice – try doing things in SQL, too, using your favorite DBMS.
• "Datalog" : "Database + Prolog", a notion coined around 1984 in the USA.

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

• Semantically: Strongly different! Set-oriented like other languages, e.g., SQL (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, no DBMS product!

• Rather uniform syntax and semantics of facts and rules

• Various different proposals for queries, updates, constraints, and schemas

• Datalog is based on the domain relational calculus (DRC), while SQL is based on tuple relational calculus (TRC) and relational algebra (RA). Datalog uses just a minimal set of logical operators:

  Conjunction and Negation

• Later in this chapter: Some more background about the three formal relational DB languages just mentioned – RA, DRC, and TRC! A bit more about Prolog will follow.
**Datalog**

\[
s(X) \leftarrow p(X,Y).
\]

\[
s(X) \leftarrow r(Y,X).
\]

\[
t(X,Y,Z) \leftarrow p(X, Y), r(Y,Z).
\]

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

**SQL**

```
CREATE VIEW s AS
(SELECT a FROM p)
UNION
(SELECT b FROM r);
```

```
CREATE VIEW t AS
SELECT a, b, c
FROM p, r
WHERE p.b = r.a,
```

```
CREATE VIEW w AS
(TABLE s)
MINUS
(TABLE q);
```

**Views** in SQL (as named, stored queries) have **rules** in Datalog as analogous counterparts. Three derived relations are defined in both cases: s, t, w. In both cases, three tables are used: p, q, r. View w depends on view s, too.
Basic Datalog Syntax on a Single Slide

Consants

**Facts**

- p(1,a).
- p(2,b).
- p(3,c).
- q(2).
- q(5).
- r(a,1).
- r(a,2).
- r(b,3).

**Variables**

- s(X) ← p(X,Y).
- s(X) ← r(Y,X).
- t(X,Y,Z) ← p(X,Y), r(Y,Z).
- w(X) ← s(X), not q(X).

**Relation Names**

- p, q, r: Base relations
- s, t, w: Derived relations

**Implicit disjunction (or)**

**Conjunction (and)**

**Negation (not)**
No Nulls in Datalog (1)

Datalog cannot accommodate null values, i.e., no empty cells in a table!

Reasons for nulls in the „SQL database“:

- Everyone except Diana is still alive.
- No ancestors for the royal couple.
- No ancestor data for spouses of royals.
- Some spouses don‘t have any title.

Therefore: „Datalog DB“ needs more than two (normalized) tables.
No Nulls in Datalog (2)

Person (SQL)

<table>
<thead>
<tr>
<th>Title</th>
<th>Name</th>
<th>Father</th>
<th>Mother</th>
<th>Birth</th>
<th>Death</th>
<th>Sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queen</td>
<td>Elizabeth II</td>
<td>Philip</td>
<td></td>
<td>1926</td>
<td>1987</td>
<td>f</td>
</tr>
<tr>
<td>Prince</td>
<td>Charles</td>
<td>Philip</td>
<td>Elizabeth II</td>
<td>1948</td>
<td></td>
<td>m</td>
</tr>
<tr>
<td>Princess Anne</td>
<td>Philip</td>
<td>Elizabeth II</td>
<td>1960</td>
<td></td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>Prince Andrew</td>
<td>Philip</td>
<td>Elizabeth II</td>
<td>1960</td>
<td></td>
<td>m</td>
<td></td>
</tr>
<tr>
<td>Prince Edward</td>
<td>Philip</td>
<td>Elizabeth II</td>
<td>1961</td>
<td></td>
<td>m</td>
<td></td>
</tr>
<tr>
<td>Princess Diana</td>
<td></td>
<td></td>
<td></td>
<td>1964</td>
<td>1987</td>
<td>f</td>
</tr>
<tr>
<td>Duchess Camilla</td>
<td></td>
<td></td>
<td></td>
<td>1947</td>
<td></td>
<td>f</td>
</tr>
<tr>
<td>Prince William Charles Diana</td>
<td>Charles</td>
<td>Diana</td>
<td>1962</td>
<td></td>
<td>m</td>
<td></td>
</tr>
<tr>
<td>Prince Henry Charles Diana</td>
<td>Charles</td>
<td>Diana</td>
<td>1964</td>
<td></td>
<td>m</td>
<td></td>
</tr>
<tr>
<td>Mark</td>
<td></td>
<td></td>
<td></td>
<td>1948</td>
<td></td>
<td>m</td>
</tr>
<tr>
<td>Sir Timothy</td>
<td></td>
<td></td>
<td></td>
<td>1950</td>
<td></td>
<td>m</td>
</tr>
<tr>
<td>Duchess Sarah</td>
<td></td>
<td></td>
<td></td>
<td>1959</td>
<td></td>
<td>f</td>
</tr>
<tr>
<td>Countess Sophie</td>
<td></td>
<td></td>
<td></td>
<td>1965</td>
<td></td>
<td>m</td>
</tr>
<tr>
<td>Peter</td>
<td>Mark</td>
<td>Anne</td>
<td></td>
<td>1977</td>
<td></td>
<td>m</td>
</tr>
<tr>
<td>Zara</td>
<td>Mark</td>
<td>Anne</td>
<td></td>
<td>1981</td>
<td></td>
<td>f</td>
</tr>
<tr>
<td>Lady Louise</td>
<td>Edward</td>
<td>Sophie</td>
<td></td>
<td>2003</td>
<td></td>
<td>f</td>
</tr>
<tr>
<td>Viscount James</td>
<td>Edward</td>
<td>Sophie</td>
<td></td>
<td>2003</td>
<td></td>
<td>m</td>
</tr>
<tr>
<td>Princess Beatrice Andrew Sarah</td>
<td>Andrew</td>
<td>Sarah</td>
<td>1998</td>
<td></td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>Princess Eugenie Andrew Sarah</td>
<td>Andrew</td>
<td>Sarah</td>
<td>1990</td>
<td></td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>Duchess Catherine</td>
<td></td>
<td></td>
<td></td>
<td>1982</td>
<td></td>
<td>m</td>
</tr>
<tr>
<td>Autumn</td>
<td></td>
<td></td>
<td></td>
<td>1970</td>
<td></td>
<td>f</td>
</tr>
<tr>
<td>Prince George William Kate</td>
<td>William</td>
<td>Kate</td>
<td>2013</td>
<td></td>
<td>m</td>
<td></td>
</tr>
<tr>
<td>Michael</td>
<td></td>
<td></td>
<td></td>
<td>1980</td>
<td></td>
<td>f</td>
</tr>
<tr>
<td>Isla</td>
<td>Peter</td>
<td>Autumn</td>
<td></td>
<td>2012</td>
<td></td>
<td>f</td>
</tr>
</tbody>
</table>

Person (Datalog)

<table>
<thead>
<tr>
<th>Name</th>
<th>Birth</th>
<th>Sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Charles</td>
<td>1948</td>
<td>m</td>
</tr>
<tr>
<td>Anne</td>
<td>1950</td>
<td>f</td>
</tr>
<tr>
<td>Andrew</td>
<td>1960</td>
<td>m</td>
</tr>
<tr>
<td>Edward</td>
<td>1964</td>
<td>m</td>
</tr>
<tr>
<td>Diana</td>
<td>1961</td>
<td>f</td>
</tr>
<tr>
<td>Camilla</td>
<td>1947</td>
<td>f</td>
</tr>
<tr>
<td>William</td>
<td>1982</td>
<td>m</td>
</tr>
<tr>
<td>Henry</td>
<td>1984</td>
<td>m</td>
</tr>
<tr>
<td>Mark</td>
<td>1940</td>
<td>m</td>
</tr>
<tr>
<td>Timothy</td>
<td>1950</td>
<td>m</td>
</tr>
<tr>
<td>Sarah</td>
<td>1959</td>
<td>f</td>
</tr>
<tr>
<td>Sophie</td>
<td>1955</td>
<td>f</td>
</tr>
<tr>
<td>Peter</td>
<td>1977</td>
<td>m</td>
</tr>
<tr>
<td>Zara</td>
<td>1981</td>
<td>f</td>
</tr>
<tr>
<td>Louise</td>
<td>2003</td>
<td>f</td>
</tr>
<tr>
<td>James</td>
<td>2007</td>
<td>m</td>
</tr>
<tr>
<td>Beatrice</td>
<td>1988</td>
<td>f</td>
</tr>
<tr>
<td>Eugenie</td>
<td>1990</td>
<td>f</td>
</tr>
<tr>
<td>Catherine</td>
<td>1992</td>
<td>f</td>
</tr>
<tr>
<td>Autumn</td>
<td>1978</td>
<td>f</td>
</tr>
<tr>
<td>Savannah</td>
<td>2010</td>
<td>f</td>
</tr>
<tr>
<td>George</td>
<td>2013</td>
<td>m</td>
</tr>
<tr>
<td>Michael</td>
<td>1997</td>
<td>m</td>
</tr>
<tr>
<td>Isla</td>
<td>2012</td>
<td>f</td>
</tr>
</tbody>
</table>

Child (Datalog)

<table>
<thead>
<tr>
<th>Name</th>
<th>Father</th>
<th>Mother</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Philip</td>
<td>Elizabeth II</td>
<td>Charles</td>
<td>Diana</td>
</tr>
<tr>
<td>Philip</td>
<td>Elizabeth II</td>
<td>Anna</td>
<td>Peter</td>
</tr>
<tr>
<td>Philip</td>
<td>Elizabeth II</td>
<td>Andrew</td>
<td>Sarah</td>
</tr>
<tr>
<td>Philip</td>
<td>Elizabeth II</td>
<td>Edward</td>
<td>Sarah</td>
</tr>
<tr>
<td>Charles</td>
<td>Diana</td>
<td>William</td>
<td>Isla</td>
</tr>
</tbody>
</table>

Death (Datalog)

<table>
<thead>
<tr>
<th>Name</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diana</td>
<td>1997</td>
</tr>
</tbody>
</table>

Three null-free tables rather than one

(Title dropped from now on)
Similar normalization required for Marriage (as most royal couples are not – yet – divorced).
CREATE VIEW Brother AS
(SELECT    P1.Name,
        P2.Name  AS Brother
FROM Person AS P1,
        Person AS P2,
WHERE P1.Mother = P2.Mother
AND       P1.Father = P2.Father
AND       P1.Name <> P2.Name
AND       P2.Sex = 'm')

In order to compare the SQL views with the corresponding Datalog rules, we first have to „translate“ view definitions to the new 5-table schema.

Person information in Datalog:
- person(Name, Birth, Sex)
- child(Father, Mother, Child)
- death(Name, Year)

Person table in SQL:
<table>
<thead>
<tr>
<th>Title</th>
<th>Name</th>
<th>Father</th>
<th>Mother</th>
<th>Birth</th>
<th>Death</th>
<th>Sex</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

old version

new version
"Brother" in Datalog

**SQL:**

```
CREATE VIEW Brother AS
(SELECT      C1.Child   AS Name,
     C2.Name  AS Brother
FROM    Child    AS C1,
          Child    AS C2,
          Person  AS  P2
WHERE   C1.Mother = C2.Mother
       AND       C1.Father = C2.Father
       AND       C1.Child = P2.Name
       AND       P2.Sex = 'm'
)
```

**Datalog:**

```
brother(N,B) ←
    child(F,M,N),
    child(F,M,B),
    person(B,_,'m'),
N <> B.
```

In SQL: Variables for rows of tables.
In Datalog: Variables for attribute values of rows.
• The basic building blocks of Datalog rules are called literals. They consist of a relation name and a list of parameters (either variables or constants), e.g., \textit{child}(F,M,N).

• \textbf{Variables} in Datalog stand for individual attribute values, i.e., elements of data types occupying a single cell in the table/view under consideration.

• Datalog doesn‘t make use of any attributes (column names), though. Columns are identified by their position within the parameter list. The order of columns matters and has to be fixed.

• Datalog rules are expressions of the form \textit{Head} \leftarrow \textit{Body}. where \textit{Head} is a literal representing name and column structure of the newly defined view (derived relation). The rule body is a conjunction of literals accessing tables or other views on which the new view depends.

• The same variable occurring in different places in a rule always represents the same value (variable binding), whereas different variables may (but don‘t have to) represent different values.

• \textbf{Underscores} represent „unnamed“ variables „filling up“ irrelevant positions in a parameter list.

\begin{verbatim}
brother(N,B) \leftarrow child(F,M,N), child(F,M,B), person(B,_,’m’), N <> B.
\end{verbatim}
CREATE VIEW Parent AS

((SELECT  Child    AS  Name,
Father    AS  Parent
FROM Child)
UNION
(SELECT Child    AS  Name,
Mother    AS  Parent
FROM Child))

CREATE VIEW Uncle AS

(SELECT    P.Name AS Name,
B.Brother   AS Uncle
FROM Parent  AS   P,
Brother AS   B
WHERE P.Name = B.Name)

New feature: underscore _ for „don’t care“ positions in parameter lists of literals

\[
\text{parent}(N,P) \leftarrow \text{child}(P,_,N).
\]

\[
\text{parent}(N,P) \leftarrow \text{child}(_,P,N).
\]

\[
\text{uncle}(N,\text{\textbf{U}}) \leftarrow
\text{parent}(N,\text{\textbf{P}}),
\text{brother}(\text{\textbf{P}},\text{\textbf{U}}).
\]
uncle(N, U) ←
  parent(N, P),
  brother(P, U).

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

brother(N, B) ←
  child(F, M, N),
  child(F, M, B),
  person(B, _, 'm'),
  N <> B.

child/3

person/3
CREATE VIEW Parent AS
  ((SELECT Child AS Name, Father AS Parent
   FROM Child)
  UNION
  (SELECT Child AS Name, Mother AS Parent
   FROM Child))

CREATE VIEW Brother AS
  (SELECT C1.Child AS Name, B.Brother AS Brother
   FROM Child AS C1,
       Child AS C2,
       Person AS P2
   WHERE C1.Mother = C2.Mother
       AND C1.Father = C2.Father
       AND C1.Child = P2.Name
       AND P2.Sex = 'm')

CREATE VIEW Uncle AS
  (SELECT P.Name AS Name, B.Brother AS Uncle
   FROM Parent AS P,
       Brother AS B
   WHERE P.Name = B.Name)
• In our initial example comparing Datalog and SQL (when declaring the same derived relations/tables) we used one rule/view containing logical negation:

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

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

• In Datalog, the Boolean operator 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 MINUS is used instead.

• Negation (regardless in which syntatic variant) potentially causes trouble in a deductive (relational) database – to be explained later. Therefore we will have to treat negation 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, 1st 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!).

\[
\text{interesting\_uncle}(N,U) \leftarrow \text{uncle}(N,U),
\text{not} \ \text{ever\_married}(U),
\text{not} \ \text{has\_child}(U).
\]
Corresponding Data over Royals DB

Interesting uncles

```plaintext
interesting_uncle(N, U) ← uncle(N, U),
not ever_married(U),
not has_child(U).
```

How to compute interesting uncles if you only have positive data at hand?
Discovering Negative Data (1)

Although we do not have data about persons who never married, or are childless . . .

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

... 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.
Discovering Negative Data (2)

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}):

```sql
CREATE VIEW Interesting_uncle AS
(SELECT * FROM Uncle WHERE Uncle NOT IN (SELECT * FROM Ever_married)
AND Uncle NOT IN (SELECT * FROM Has_child))
```

A formulation using \texttt{MINUS} instead (and no \texttt{WHERE} part) works, too, as previously.

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

Why not simply use one rule rather than three?

Don’t use this in Datalog!

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!
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).

These arrows are not referring to the family tree, but represent dependency of relations on each other!

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

© 2018 Prof. Dr. Rainer Manthey
Intelligent Information Systems
Traversals 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:**

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

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

**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:

\[
\text{ancestor}(X,Y, \mathbf{1}) \leftarrow \text{parent}(X,Y).
\]

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

\[
\text{descendant}(X,Y, \mathbf{I}) \leftarrow \text{ancestor}(Y,X, \mathbf{I}).
\]
Computing Royal Ancestors Recursively (1)

\[
\text{ancestor}(X,Y,1) \leftarrow \text{parent}(X,Y).
\]

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

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.
Computing Royal Ancestors Recursively (2)

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

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

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

Similarly in iteration 3:
Combining generation 2 facts with parent facts.

Nothing new in iteration 4: stop!.
Computing Royal Ancestors Recursively (4)

ancestor(X, Y) ← parent(X, Y).
ancestor(X, Y) ← parent(X, Z), ancestor(Z, Y).

Ancestor

Ancestor

Ancestor

Ancestor

Ancestor

Ancestor

Ancestor

Ancestor

Ancestor

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.

(Male form only, due to space limitations)
The concept "relative of" can be defined via several Datalog rules, too, using the concepts ancestor and descendant introduced before (to be discussed in the exercises):

- **in the main line**
  - relative(X, Y, Degree) ← ancestor(X, Y, Degree).
  - relative(X, Y, Degree) ← descendant(X, Y, Degree).

- **in the sideline**
  - relative(X, Y, Degree) ← ancestor(Z, X, Degree1), ancestor(Z, Y, Degree2), not ancestor(X, Y), not ancestor(Y, X), not has_younger_common_anc(X,Y,Z), X ≠ Y, Degree = Degree1 + Degree2.

- **has_younger_common_anc(X,Y,Z)** ← ancestor(Z1, X, _), ancestor(Z1, X, _), ancestor(Z, Z1, _).
Now for Some Theory!

2.1 Datalog and SQL: Learning by Doing
2.2 **Datalog: Syntax and (Basic) Semantics**
2.3 From SQL to Datalog (and Back)
About this Section

• After an intuitive, example-based introduction to the most important principles of Datalog in 2.1, 2.2 will look at the features and conventions of this language from a more abstract, systematic point of view.

• The introduction will not be a formal one, however, even though it is no problem to come along with a formal grammar for the syntax of each construct.

• Semantics is more problematic and will be dealt with in chapter 3 (formally, at least in part).

• Many of the introductory slides will just repeat „officially“ what was already mentioned before – but there will be key features (such as safety, CWA, or negation-as-failure) which will be discussed more in-depth as they are a bit intricate.

• Altogether, this is a section for your own reading more than for lengthy oral presentation within the lecture.

• Take this serious nevertheless – the details of the language will be relevant for the rest of the semester (including the exam).
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>
• **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%

   („Inherited“ from conventions in most Prolog systems)

   ... 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
Tables

conventional table
(à la SQL)

<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>

Table (relation) as a set of facts in Datalog

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 ).
Deductive Rules in Datalog: Structure

Rule head

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

Rule body

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

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 ).

Derived relation

Base relation

Implication symbol as in mathematical logic
Another "syntactical tradition" from the Prolog world:
Facts and rules are always terminated with a **dot**!

```
population( 'Berlin', 3399 ).
```

```
population(City, Inhabitants) ←
  city(City, Phone, Car, Inhabitants).
```

(As **Datalog** is **not standardized** – like, e.g., **SQL** – such syntactic rules are quite often neglected by authors following a tradition of their own. Take care!)
• 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.

<table>
<thead>
<tr>
<th>positive literals</th>
<th>negative literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(X,Y)</td>
<td>not p(X,Y)</td>
</tr>
<tr>
<td>q(a, Y, 1)</td>
<td>not q(a,Y,1)</td>
</tr>
<tr>
<td>r(a,b)</td>
<td>not r(a,b)</td>
</tr>
</tbody>
</table>

• 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.
• Literals play a **double role** in Datalog:
  - In facts and rule heads:
    For **asserting** confirmed or assumed data.
  - In rule bodies (and later on in queries):
    For **finding** confirmed or assumed data.

• A fact in a deductive database asserts, **that** something is the case:
  
  \[
  \text{child_of('William', 'Diana', 'Charles').}
  \]

• This is true for derived facts as well, thus rule heads play an asserting role, too:
  
  \[
  \text{father_of(Y, Z) } \leftarrow \text{child_of(Z, X, Y).}
  \]

• Literals in role bodies, however, serve as questions **if** something is the case resp. **for which** variable substitutions something is the case:
  
  \[
  \text{father_of(Y, Z) } \leftarrow \text{child_of(Z, X, Y).}
  \]

• In order to derive father_of-facts with this rule, it is necessary to **find** child_of-facts and to transfer variable bindings found to the head literal, in order to **assert** new facts.
Meaning of Literals

- Literals on „asserting“ positions – i.e., in facts and rule heads – „mean themselves“.

- Literals in „querying“ positions – i.e., in particular in rule bodies – get their meaning via an evaluation over a certain set of facts only, e.g.:

  ```
  father_of(Y, Z) ← child_of(Z, X, Y).
  ```

  child_of('Charles', 'Elizabeth', 'Philip').
  child_of('Anne', 'Elizabeth', 'Philip').
  child_of('Andrew', 'Elizabeth', 'Philip').
  child_of('Edward', 'Elizabeth', 'Philip').

  - The body literal represents an (atomic) query:
    For which (X,Y,Z)-combination is the resulting fact true?

  - The expected answer is the set of all variable substitutions that can be obtained by evaluating the literal over the given set of facts.
Application of Deductive Rules: 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

Base facts or previously derived facts
• 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 . . .

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

. . . 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?

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

- **Later in this lecture**: Unsafe rules will be admitted though under certain contraints!

- **But for now**: All rules are assumed to be safe – **no unsafe rules**!
**Conjunctive Rules**

- **is_a_capital( 'München', 'M' ).**
- **capital_of( 'Bavaria', 'München' ).**
- **city( 'München', 089, 'M', 1189 ).**

### Concatenation via identical variables
- **is_a_capital (City, Car) ← capital_of(State, City) , city(City, Phone, Car, Inhabitants).**

### Comma as ∧-operator
- **Comma as ∧-operator**

### Substitution with identical values
- **Substitution with identical values**
Derived Relations may be defined by more than one rule:

\[
\begin{align*}
\text{european_city}(X) & \leftarrow \text{situated_in}(X, \text{'Denmark'}) . \\
\text{european_city}(X) & \leftarrow \text{situated_in}(X, \text{'Germany'}) . \\
\text{european_city}(X) & \leftarrow \text{situated_in}(X, \text{'Russia'}) . \\
\ldots
\end{align*}
\]

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).
Order Independence 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:

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

• 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.
∃-Position for Local Variables: Motivation

• The convention
  „Implicit ∃-quantifiers are always assumed to stand in front of all literals!“ seems to be unnecessary on first glance. For this reason, it is worthwhile to discuss alternatives by means of a more complex example:

\[ p(X) \leftarrow q(X,Z), r(Z,X, Y), s(Y,W). \]

\[ \exists \quad Y,Z,W \]

• The only true alternative would be to keep the „scope“ (range of validity) of quantifiers as small as possible and thus to make them individually different:

\[ p(X) \leftarrow q(X,Z), r(Z,X, Y), s(Y,W). \]

\[ \exists \quad Z \]
\[ \exists \quad W \]
\[ \exists \quad Y \]

Disadvantage:
• Considerably more complex
• Different quantifier structure for each reordering of the rule body
• 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.

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

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

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

"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 (the „true name“ of which we do not know):

  is_a_capital(\texttt{City}, \texttt{Car}) \leftarrow
  \texttt{capital\_of ( \_ , City) , city(City, \_ , Car, \_ ).}

- Each of these variables has **its own implicit quantifier**. All these quantifiers is placed (as usual) at the beginning of the rule body (see above).
Hierarchical Dependencies

Important in particular for constructing terminological hierarchies:

**Derived** relations 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.:**

\[
p(X,Y) \leftarrow q(X,Y), r(Y, X).
\]

\[
q(X,Y) \leftarrow s(X, Y, Z).
\]

\[
r(Y, X) \leftarrow t(X,Y), \text{not } w(Y).
\]

Corresponding **dependency graph:**

```
   p
  /\   /
 q  r
/\  /\  \
 s  t  w
```

**not**
• 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")

e.g.:

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!
Instances of Datalog Rules

• By stepwise replacement of variables by constants in a rule, various instances of this rule can be obtained:

\[
p(X, Y) \leftarrow q(X, Z), r(Z, Y).
\]

- \( p(a, Y) \leftarrow q(a, Z), r(Z, Y). \)
- \( p(b, 1) \leftarrow q(b, c), r(c, 1). \)

"Ground instance"
(All variables have been replaced by constants.)

• In each step, consistent replacements are admissible only, i.e., different occurrences of the same variable are always replaced by the same constant.

\[
p(a, Y) \leftarrow q(b, Z), r(Z, Y).
\]

Not an instance!
• 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 not 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!
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:

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.

The complement of the DB is never explicitly computed or even stored (for efficiency reasons)!

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
Negation in Datalog

- 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: and not

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

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

Production \hspace{2cm} Test
Evaluating Negative Literals

normal_city( 'Köln', 'K' ).

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

City/'Köln'
Car/'K'

is_a_capital('Köln', 'K') ?
? yes!

City/ 'München', 089, 'M', 1189).

is_a_capital('München', 'M' ).
is_a_capital('Düsseldorf', 'D').

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 not F, . . .
  • . . . try to answer its 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)': 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:

\[
\exists \text{State:} \quad \text{normal\_city}(\text{City}, \text{Car}) \leftarrow \text{city}(\text{City}, _, \text{Car}, _), \not \text{capital\_of}(\text{State}, \text{City}).
\]

Auxiliary relation

\[
\exists \text{State:} \quad \text{is\_a\_capital}(\text{City}) \leftarrow \text{capital\_of}(\text{State}, \text{City}).
\]
• 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(MatNr)`, `exams(MatNr,ModNr,Result)`, and `modules(ModNr,Status)`. Further assume that exam results are either *pass* or *fail*, and that module status is either *mandatory* 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.
Simulating Universal Quantifiers (2)

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:

\[
\text{successful}(S) \leftarrow \text{students}(S) \text{ and } \neg \exists M: \text{modules}(M,'m') \text{ and } \neg \text{exams}(S,M,'pass')).
\]

- 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:

\[
\text{successful}(S) \leftarrow \\
\text{students}(S), \neg \text{has_failed_module}(S).
\]

\[
\text{has_failed_module}(S) \leftarrow \\
\text{students}(S), \text{modules}(M,'m'), \neg \text{exams}(S,M,'pass')).
\]
• Doesn’t belong to the „core“ of Datalog, by unavoidable in practice:

• 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.

  \[
  X > Y \quad X \leq 1 \quad \text{not} \quad a = b
  \]
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:

  ⇒ 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!
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:

Table → Aggregate → Function value
Aggregates in SQL: Reminder

This is how aggregates work in SQL:

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

### TABLE

<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

```
Name | Rank | Age
--- | ---- | ----
Jim  | C4   | 43  
Lisa | C4   | 39  
Ken  | C4   | 57  
Lisa | C4   | 39  
John | C3   | 33  
Eva  | C3   | 36  
Tom  | C2   | 32  
```

### AVG

<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.:

```
avg_salary(Dept, AvgS) ← AvgS = avg(Salary, Dept, employee(E, Dept, Salary))
```

• There are special ternary aggregate functions: avg, sum, max, min, card.
  
  (We prefer card(inality) rather than count.)

• The 1\textsuperscript{st} parameter of each aggregate term is the variable to be aggregated about.

• The 2\textsuperscript{nd} parameter is the 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 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
A Tale of Two Languages

2.1 Datalog and SQL: Learning by Doing
2.2 Datalog: Syntax and (Basic) Semantics
2.3 From SQL to Datalog (and Back)
Section 2.3 plays an important role in various respects:

- It tries to link the („academic“) key language of this lecture, Datalog, to the key language in the „professional“ DB world, SQL, in order to make sure that you know about the relevance of our topics for the „real world“.
- It tries to provide you with a solid background on the theoretical basis of the languages covered, a background that is relevant for more recent approaches to IS/DB models and formalisms, too (such as XML and RDF, e.g.).

At the same time, this section – extending just to the next lecture this week – is not able to provide you with a more detailed introduction to the languages addressed, in particular SQL. Effort of your own is needed, at least for making sure you leave the university with skills you can directly use in practice.

For our exam expectations, however, we try to treat everything you need explicitly and deep enough.
The order in which topics will be presented this week is not the one which will be chosen finally, due to „context reasons“.

In particular, the foundations ought to be discussed first, not last (as we will have to do in the lecture).

SQL basics would be necessary as second topic in 2.3, as Datalog has been well introduced enough. This topic will also appear in a bit more detail and in a different position within the slide order later.

Even though the „Datalog to SQL“ transformation is the more important one (wrt „exporting“ results from this lecture to the SQL „world“) we will start the other way round and address “SQL to Datalog” first, again for “context reasons”.

We will „call“ this part SQL2Datalog in the headings. Slides are from last year (and thus directly „re-usable“).
• Every result (to be) presented during this lecture in the context of Datalog can be transferred to SQL – the only exception concerning unstratifiable rules to be discussed in 3.3.

• We will discuss translation of SQL views (vice versa) into Datalog rules first in this section. The examples chosen can be easily generalized.

• A core sublanguage of SQL is sufficient for mapping into Datalog (and back). Core SQL offers the following operators:
  • SELECT-FROM-WHERE queries (without nesting in SELECT and FROM)
  • JOIN and its variants are omitted: They can be expressed using „normal“ FROM and join conditions in the WHERE-part.
  • UNION is the only operator needed for forming complex queries. OR is not needed, just AND occurs in the WHERE-part.
  • Nested subqueries (after EXISTS with and without NOT) are sufficient for expressing MINUS and INTERSECT).

• Transformation in the other direction (Datalog into SQL) uses the same techniques in principle – two slides on this issue will be discussed at the end of this chapter.
CREATE VIEW p AS

SELECT A
FROM t
WHERE B>0 AND C=D AND NOT D=E

Direct translation into Datalog rule:

\[ p(X) \leftarrow t(X, Y, Z, V, W), Y>0, Z=V, \text{not } V=W. \]

More compact rule by expressing (consequences of) equation using the same variable:

\[ p(X) \leftarrow t(X, Y, Z, Z, W), Y>0, \text{not } Z=W. \]
CREATE VIEW p AS

\[
\text{SELECT A} \\
\text{FROM t} \\
\text{WHERE B>0 AND C=D AND NOT D=E}
\]

**Implicit tuple variable in query:**

\[
\text{SELECT x.A} \\
\text{FROM t AS x} \\
\text{WHERE x.B>0 AND x.C=x.D AND NOT x.D=x.E}
\]

- \(x\) bound to \(t\)-tuples one after the other
- Attributes are functions „extracting“ components from current value of \(x\).
- Function application expressed in postfix notation

\[p(X) \leftarrow t(X, Y, Z, V, W), Y>0, Z=V, \text{not } V=W.\]

**TRC: Tuple Relational Calculus**

**DRC: Domain Relational Calculus**

- Variables represent components of the same tuple in \(t\)
- „same tuple in \(t\)“: expressed by common \(t\)-literal
CREATE VIEW p AS

SELECT A
FROM r, s
WHERE r.B=s.A

Same translation idea:

\[ p(X) \leftarrow r(X, Y, Z), s(V, W), Y=V. \]

Compactification by multiple variable occurrence and anonymous variables:

\[ p(X) \leftarrow r(X, Y, _), s(Y, _). \]

New tables:
- \( r(A, B, C) \)
- \( s(A, B) \)

Variant with double occurrence of \( s \) and aliasing:

CREATE VIEW p AS

SELECT A
FROM s AS s1, s AS s2
WHERE s1.B=s2.A

\[ p(X) \leftarrow s(X, Y), s(Y, _). \]
CREATE VIEW p AS

\[
\begin{align*}
\text{SELECT} & \quad A \\
\text{FROM} & \quad r \ JOIN s \ ON \ r.B = s.A
\end{align*}
\]

\[
p(X) \leftarrow r(X, Y, _), s(Y, _).
\]

JOIN queries are dispensable in SQL, they are just syntactic alternatives for products queries with conditions in the WHERE part:

\[
\begin{align*}
\text{SELECT} & \quad A \\
\text{FROM} & \quad r, s \\
\text{WHERE} & \quad r.B = s.A
\end{align*}
\]

Datalog doesn't offer any such special „luxury“ notations: Same translation!
CREATE VIEW p AS

\[
\text{SELECT } A \\
\text{FROM } r \\
\text{WHERE } B=C
\]

UNION

\[
\text{SELECT } A \\
\text{FROM } s \\
\text{WHERE } B>0
\]

Each subquery in a UNION query is turned into a rule of its own, defining the same relation:

\[
p(X) \leftarrow r(X, Y, Y).
p(X) \leftarrow s(X, Z), Z>0.
\]

Variant: Y instead of Z in 2nd rule (Ys are different in different rules!)

\[
p(X) \leftarrow r(X, Y, Y).
p(X) \leftarrow s(X, Y), Y>0.
\]
CREATE VIEW p AS

SELECT A
FROM t
WHERE B>0 OR C=D

OR in WHERE parts is dispensable, UNION serves the same purpose!

CREATE VIEW p AS

SELECT A
FROM t
WHERE C=D

CREATE VIEW p AS

SELECT A
FROM t
WHERE B>0

UNION

p(X) ← t(X, _, Z, Z, _).
p(X) ← t(X, Y, _, _, _), Y>0.
SQL2Datalog (3b): UNION Cannot Always be Turned into OR in WHERE-part

CREATE VIEW p AS

\[
\text{SELECT A} \\
\text{FROM } r \\
\text{WHERE } B=C \\
\text{UNION} \\
\text{SELECT A} \\
\text{FROM } s \\
\text{WHERE } B>0
\]

Both tables in FROM part generate product of tables!

\[
\text{SELECT A} \\
\text{FROM } r, s \\
\text{WHERE } B>0 \text{ OR } B=C
\]

UNION is more general than OR as it applies to inhomogeneous input as well (two different tables in sub-queries):

In such a case: No transformation using OR in a single query is possible!

\[
p(X) \leftarrow t(X, \_, Z, Z, \_). \\
p(X) \leftarrow t(X, Y, \_, \_, \_), Y>0.
\]
CREATE VIEW p AS

SELECT A
FROM t
WHERE B>0 AND (C=D OR D=E)

1st step: Un-nesting using laws of
Boolean algebra til OR is outermost

SELECT A
FROM t
WHERE (B>0 AND C=D) OR (B>0 AND D=E)

2nd step: Expressing OR by UNION
(as before)

UNION

SELECT A
FROM t
WHERE B>0 AND C=D

SELECT A
FROM t
WHERE B>0 AND D=E

p(X) ← t(X, Y, Z, Z, _). Y>0.
p(X) ← t(X, Y, _, Z, Z), Y>0.

CREATE VIEW p AS

SELECT A
FROM t
WHERE B>0 AND C=D

SELECT A
FROM t
WHERE B>0 AND D=E

p(X) ← t(X, Y, Z, Z, _). Y>0.
p(X) ← t(X, Y, _, Z, Z), Y>0.
CREATE VIEW p AS

```
SELECT A
FROM r
WHERE B=C
```

MINUS

```
SELECT A
FROM s
WHERE B>0
```

```
SELECT A
FROM r
WHERE B=C AND NOT EXISTS
(SELECT *
FROM s
WHERE B>0 AND s.A=r.A)
```

```
p(X) ← r(X, Y, Y), not s'(X)
s'(X) ← s(X,Z), Z>0.
```

MINUS is dispensable, too, as NOT EXISTS serves the same purpose (and shows clearer which subquery plays „generator role“ and which one „filter role“)

Auxiliary rule needed for embedded subquery in order to guarantee safe negation.
CREATE VIEW p AS

\[
\text{SELECT A} \\
\text{FROM } r \\
\text{WHERE } B=C
\]

\[
\text{INTERSECT}
\]

\[
\text{SELECT A} \\
\text{FROM } s \\
\text{WHERE } B>0
\]

\[
\text{SELECT A} \\
\text{FROM } r \\
\text{WHERE } B=C \text{ AND EXISTS (}
\text{SELECT *} \\
\text{FROM } s \\
\text{WHERE } B>0 \text{ AND } s.A=r.A)
\]

\[
p(X) \leftarrow r(X, Y, Y), s'(X).
\]

\[
s'(X) \leftarrow s(X,Z), Z>0.
\]

Auxiliary relation \( s' \) instead of nesting in Datalog.
CREATE VIEW p AS

SELECT A
FROM r
WHERE B=C

INTERSECT

SELECT A
FROM s
WHERE B>0

As opposed to NOT EXISTS: EXISTS queries can be „folded back“ into a single-level query!

SELECT r.A
FROM r, s
WHERE r.B = r.C AND s.B > 0
AND s.A = r.A

p(X) ← r(X, Y, Y), s(X, Z), Z > 0.

Corresponding effect in Datalog: No auxiliary rule necessary as not is missing!
CREATE RECURSIVE VIEW p AS
( (SELECT *
   FROM s)
UNION
(SELECT s.A, p.B
   FROM s, p
   WHERE s.B=p.A ) )

\[ p(X,Y) \leftarrow s(X,Y). \]
\[ p(X,Y) \leftarrow s(X,Z), p(Z,Y). \]

SQL has recursive views, like Datalog having recursive rules. Therefore, every recursive view in SQL can be equivalently translated into a set of recursive Datalog rules (but not always vice versa).

There are certain restrictions remaining in SQL, though, most notable:
- linear recursion
- stratifiable rule sets (see chapter 3)