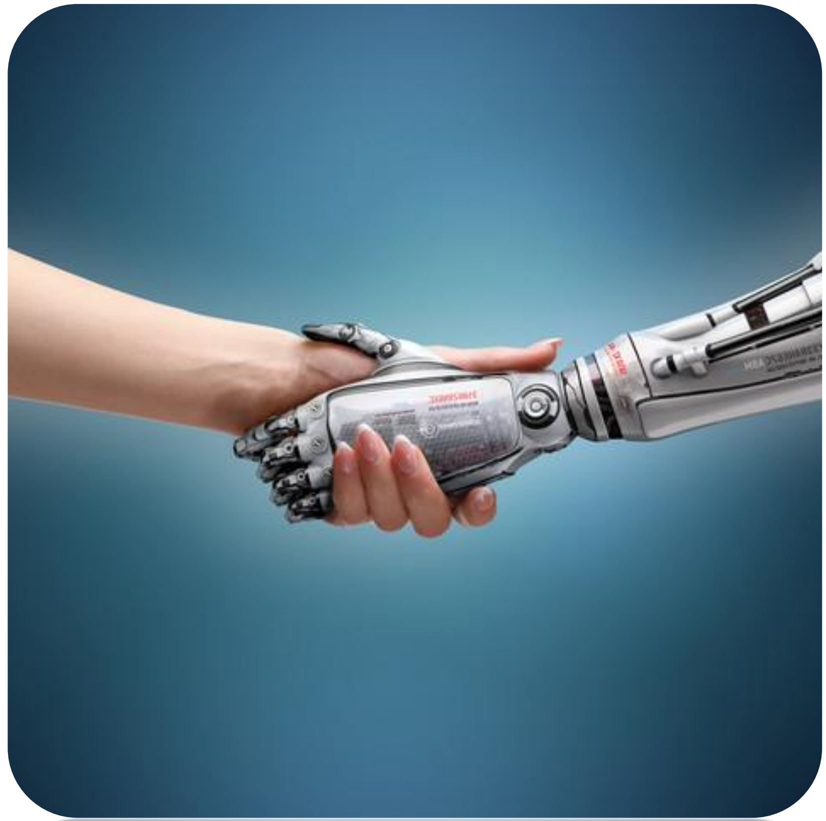
Zurich University
of Applied Sciences

Artificial Intelligence
V06b: Datalog

Recap: propositional & first-order logic
Reasoning in databases – an example
Datalog



Based on material by
- Stuart Russell, UC Berkeley
- Bill Howe, U Washington
- Kevin Leyton-Brown, U British Columbia

Zurich University of Applied Sciences and Arts
InIT Institute of Applied Information Technology (stdm)

Zurich University
of Applied Sciences

Educational objectives

- **Remember** the **syntax** & **semantics** of **Datalog** and its derivation from predicate- and first-order logic
- **Explain** **how** the **special cut of** Datalog‘s **features** out of different logics **produces** an **expressive yet fast** system
- **Formulate** intelligent **queries** over databases **using Datalog**

“In which we notice that the world is blessed with many objects, some of which are related to other objects, and in which we endeavor to reason about them.”

➔ Reading: AIMA, ch. 8 [+ ch. 9] (ch. 7.5.3; 9.3.1 covered here)
(ch. 8.4-8.5; 9.3 related material)

1. RECAP: PROPOSITIONAL & FIRST-ORDER LOGIC

Prerequisite 1: Propositional logic
(DE “Aussagenlogik”)

Reasoning over (unrelated) **facts**
- The **simplest of all logics** to illustrate basic ideas

Syntax
- If S is a sentence, $\neg S$ is a sentence (negation)
- If S_1 and S_2 are sentences, $S_1 \wedge S_2$ is a sentence (conjunction, “and”)
- If S_1 and S_2 are sentences, $S_1 \vee S_2$ is a sentence (disjunction, “or”)
- If S_1 and S_2 are sentences, $S_1 \Rightarrow S_2$ is a sentence (implication)
- If S_1 and S_2 are sentences, $S_1 \Leftrightarrow S_2$ is a sentence (biconditional)

Semantics (**rules for evaluating truth** with respect to a model m)

$\neg S$	is true iff	S	is false		
$S_1 \wedge S_2$	is true iff	S_1	is true **and**	S_2	is true
$S_1 \vee S_2$	is true iff	S_1	is true **or**	S_2	is true
$S_1 \Rightarrow S_2$	is false iff	S_1	is true **and**	S_2	is false
$S_1 \Leftrightarrow S_2$	is true iff	$S_1 \Rightarrow S_2$	is true **and**	$S_2 \Rightarrow S_1$	is true

iff

The logical implication $S_1 \Rightarrow S_2$ (a.k.a. rule: “S_2 *if* S_1 *is true*”) shows paradox behavior when interpreted in a colloquial way:
- *“**if** I teach AI **then** the earth is a sphere”* is formally true regardless of meaning.

But the definition makes sense:
- *“**if** it is raining **then** the street gets wet”* has to be true (as a rule) regardless of if it is raining (there might be other reasons for a wet street).

See it as if saying *“**if** S1 is true **then** I claim S2 to be true as well; **else**, I make no claim”*.

Prerequisite 2: First-order logic
(FOL, DE "Prädikatenlogik 1. Stufe")

Only in higher-order logics do predicates have other predicates (or functions) as parameters

Pros and cons of propositional logic (as compared to atomic knowledge representation)

- Declarative: pieces of syntax correspond to facts
- Allows partial/disjunctive/negated information (unlike most data structures and databases)
- Compositional: meaning of $B_{1,1} \wedge P_{1,2}$ is derived from meaning of $B_{1,1}$ and of $P_{1,2}$
- Meaning is context-independent (unlike natural language, where meaning depends on context)
- Very limited expressive power (unlike natural language)
  → E.g., cannot say *"pits cause breezes in adjacent squares"* except by **one sentence for each square**!
  → It is useful to view the world as consisting of objects and relationships between them

Much greater expressiveness of FOL (like natural language)

- **Quantifiable variables** over non-logical objects (quantifiers ∀, ∃, ∄)
- **Objects**: people, houses, numbers, theories, Ronald McDonald, colors, soccer matches, wars, centuries, … Assert that the relationship exists
- **Relations** (predicates): red, round, bogus, prime, multistoried, brother of, bigger than, inside, part of, has color, occurred after, owns, comes between, …
  - **Functions**: father of, best friend, third inning of, one more than, end of, …

A function is a relation **with only one "value"** for any given "parameter"/input

Exercise: Pen&paper logic (contd.)
→ see P03b

Following Russell & Norvig's finding that *"a student of AI must develop a talent for working with logical notation"* [AIMA, p. 290], this is to get you acquainted with formulating and manipulating known facts in logical notation, and to do inference to arrive at new conclusions.

Get together in teams of 2-3 and collectively solve the following exercises from P03b using pen, paper and the previous slides. Distribute the work amongst you group and make sure to explain each result to every group member.

- 2.2 – formulating sentences in first-order logic
- 2.3 – formulating sentences in first-order logic
- 3.2 – inference in first-order logic

Prepare to explain your findings to the class.

→ See also definitions in the appendix

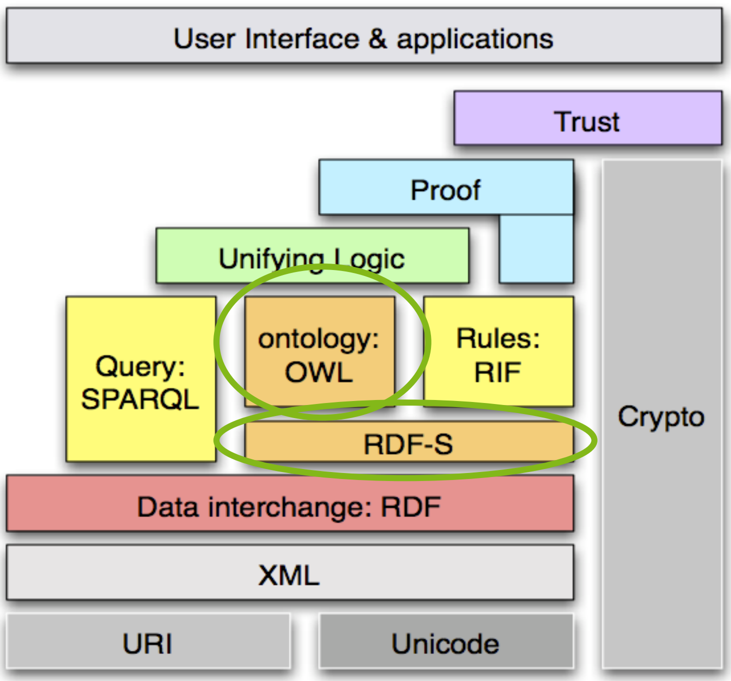
2. REASONING IN DATABASES – AN EXAMPLE

Inspired by Bill Howe's «Introduction to Data Science», lecture 9
Coursera / University of Washington

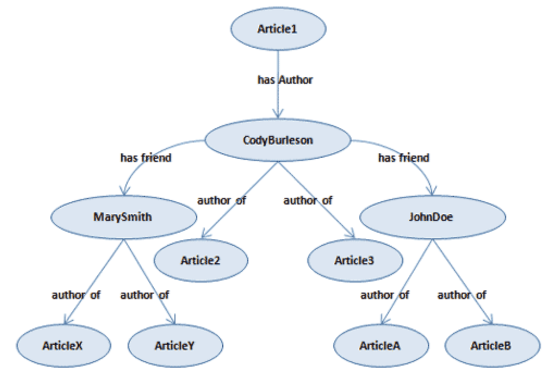
Prerequisite: Storing knowledge in graphs

based on Jana Koehler's "DB & SemWeb: Subsumption in OWL-DL", HSLU 2016

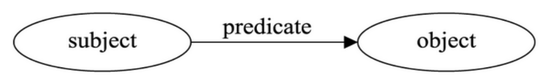
Semantic web technology stack



Implementing an ontology (a graph)…



…using triples



…in a database

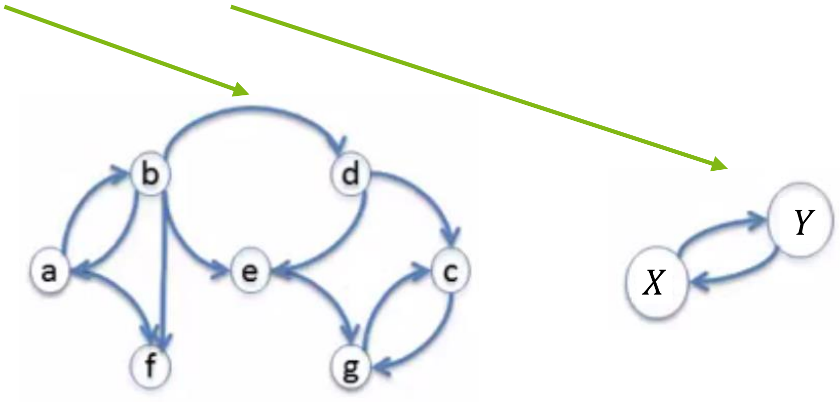
Type of	Subject	Predicate	Object
Literal	http://dbpedia.org/resource/Thessaloniki	hasName	"Thessaloniki"
Literal	http://dbpedia.org/resource/Thessaloniki	hasPopulation	363,987
Literal	http://dbpedia.org/resource/Aristotle_University	estabishedIn	1925
Literal	http://dbpedia.org/resource/Aristotle_University	hasName	"Aristotle University"
RDF	http://dbpedia.org/resource/Aristotle_University	locatedIn	http://dbpedia.org/resource/Thessaloniki

RDF Graph: A collection of five triples of various types

Problem: We're interested in pattern matching
…in graphs such as records in relational databases

Task

- For a given graph and pattern, find all instances of the pattern

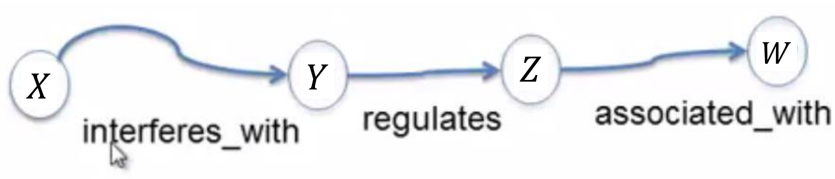


- Results:
  - $X = a, Y = b; \quad X = b, Y = a$
  - $X = g, Y = c; \quad X = c, Y = g$

Example: Adverse drug reaction research

Given a graph with edge labels
- Drug X interferes with drug Y
- Drug Y regulates the expression of gene Z
- Gene Z is associated with disease W

…find drugs that interfere with another drug involved in the treatment of a disease

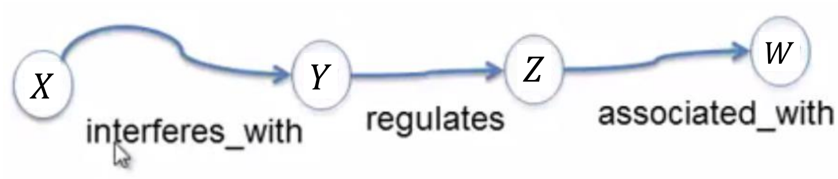


A Datalog solution

Datalog – a pattern expression language for DB queries, based on logic

- Assuming a relation $r(subject, predicate, object)$ and **pseudo syntax**

```
result(X) <=
  r(X, interferes_with, Y) &
  r(Y, regulates, Z) &
  r(Z, associated_with, W)
```



- Assuming relations $interferesWith(drug1, drug2)$, $regulates(drug, gene)$ and $associatedWith(gene, disease)$:

```
result(X) <=
  interferesWith(X, Y) &
  regulates(Y, Z) &
  associatedWith(Z, W)
```

Example: PRISM-like dragnet investigation

https://www.bloomberg.com/news/articles/2011-11-22/palantir-the-war-on-terrors-secret-weapon

«In October, a foreign national named Mike ***Fikri purchased*** *a* ***one-way plane ticket*** *from Cairo to Miami, where he rented an apartment»*

```
+ boughtFlight('Fikri', 'Cairo', 'Miami', 'oneway', 2016-10-4)
```

«Over the next few weeks, he'd made a number of ***large withdrawals*** *from a Russian bank account and placed* ***repeated calls*** *to a few people in Syria»*

```
+ withdrawal('Fikri', 5000, 'some bank', 2016-11-2)

+ withdrawal('Fikri', 2000, 'some bank', 2016-11-21)

…
```

«More recently, he ***rented a truck****, drove to Orlando, and* ***visited Disney World*** *himself»*

```
+ rented('Fikiri', 'truck', 'Miami', 'Orlando', 2017-01-30)
```

PRISM modeled in Datalog
Rules to reason over the just stated facts

```
flag(Person, 1, Date) <= boughtFlight(Person, Origin, Destination, Oneway, Date) &
                         flaggedAirport(Origin) &
                         usAirports(Destination) &
                         Oneway='oneway'

foreignWithdrawal(Person, sum(Amount), Date) <= withdrawal(Person, Amount, Bank, Date) &
                                               foreignBank(Bank) &
                                               Amount > 1000

flag(Person, 1, Date) <= foreignWithdrawal(Person, Tot_amount, Date) & Tot_amount > 10000

flag(Person, 1, Date) <= rented(Person, Vehicle, Origin, Dest, Date) & importantLocation(Dest)

…

totalflags(Person, sum(Flag), min(Date), max(Date)) <= flag(Person, Flag, Date)


alert(Person, Flag_cnt, Min_date, Max_date) <= totalflags(Person, Flag_cnt, Min_date, Max_date) &
                                               Max_date - Min_date < 10 & #days
                                               Flag_cnt > 3
```

Strengths of a high-level logical language

Who has contacted whom, when?

```
contacted(Person1, Person2, Time) <= email(Person1, Person2, Message, Time,…)
contacted(Person1, Person2, Time) <= called(Time, Voicemail, Person1, Person2, …)
contacted(Person1, Person2, Time) <= text_message(Time, Message, Person1, Person2, …)
```

The data probably comes from a lot of different systems (RDBMS, triple-store, files on Hadoop, …), but **syntactic integration doesn't take a lot of work**

Who could have known before January 30 that *X* was going to happen?

```
knew('Smith')

knew(Person2) <=
  knew(Person1) &
  contacted(Person1, Person2, Message, Time) &
  Time < 2017-01-30

knew(Person2) <=
  knew(Person1) &
  met_with(Person1, Person2, Time) &
  Time < 2017-01-30
```

Self-reference / **recursion**: not possible in standard SQL (though certain DBMS implement it)

3. DATALOG

Datalog - A relevant subset of FOL

Decision problem: A question (e.g. *"is a sentence of FOL true?"*) is decidable if an **efficient algorithm** exists that can and will return the answer (yes/no) in a finite number of steps.

Background

- Full FOL is very expressive, but **not** decidable in general
- Thus: Fallback to first-order definite clauses: *"∧" of unnegated terms ⟹ unnegated term* (more precisely, Horn clauses: also valid without the implication)
  - Some modifications (for efficient evaluation):
    - **Variables** in the head **also** appear **in the body** of a clause
    - Under certain conditions, up to **one negated term** in the body is allowed ("stratified negation")
    - Usually **no function**al symbols (not true in pyDatalog)
- Can represent the type of **knowledge** typically **found in** relational **databases**
- Still powerful (allows recursion!), but **not Turing-complete**

Datalog fundamentals

- **Clause**: is either an atomic symbol (**fact**) **or** of the form $\underbrace{\alpha}_{\text{head}} \Leftarrow \underbrace{\beta_1 \land \cdots \land \beta_m}_{\text{body}}$ (**rule**) (with atoms α, β_i)
- **Atom**: has either the form p or $p(t_1, \ldots, t_n)$, (with predicate p and terms t_i) → e.g., $p(X)$, $teaches(stadelmann, ai)$
  - **Predicate** symbol: starts with lower-case letter → e.g., p, $teaches$
  - **Term**: is either a variable or a constant
    - **Variable**: starts with upper-case letter → e.g., X, $Person1$
    - **Constant**: starts with lower-case letter or is a sequence of digits → e.g., 5, $stadelmann$, ai
- **Knowledge base**: a set of clauses

Example: Converting measures with pyDatalog
Here pure Python without DB connection

```
from pyDatalog import pyDatalog

#create terms
pyDatalog.create_terms('scale', 'A, B, C, V', 'conv') #rather atoms (terms and predicates)

#some facts (atoms, here specifically functional predicates)
scale['meter', 'inch'] = 39.3700787
scale['mile',  'inch'] = 63360.0
scale['feet',  'inch'] = 12.0

#some rules (these make it powerful: e.g. the 1. one that computes an arbitrary conversion path via recursion)
scale[A, B]   = scale[A, C] * scale[C, B] #adding transitivity
scale[A, B]   = 1/scale[B, A]
conv[V, A, B] = V * scale[A, B]

#some queries
print(scale['inch', 'meter'] == V)
print(scale['mile', 'meter'] == V)

print(conv[3, 'mile', 'meter'] == V) #note that we never explicitly defined how to convert miles to meters
print(conv[1, 'meter', 'feet'] == V)
```

Source: https://mcturra2000.wordpress.com/2014/09/14/logic-programming-example-unit-conversion-using-datalog/
Installing pyDatalog: `pip install pyDatalog`
See also: https://sites.google.com/site/pydatalog/home

Inference in Datalog

Foundation: Modus Ponens
- An inference rule known since antiquity: «If $\alpha \Rightarrow \beta$ and $\alpha == true$, then $\beta == true$»
- Also known as implication elimination

Example: Your new pet «Fritz» croaks and eats flies; **is it green?** (see https://en.wikipedia.org/wiki/Backward_chaining)
- Facts:
  - $croakes(fritz)$
  - $eatsFlies(fritz)$
- Rules:
  - $croakes(X) \wedge eatsFlies(X) \Rightarrow frog(X)$
  - $chirps(X) \wedge sings(X) \Rightarrow canary(X)$
  - $frog(X) \Rightarrow green(X)$
  - $canary(X) \Rightarrow yellow(X)$

$$\frac{P \Rightarrow Q, P}{Q}$$

Logic notation for the Modus Ponens rule

2 ways of answering this

Applying Modus Ponens **forward**

- **Data-driven**: start from true facts → use rules to derive new true facts → eventually arrive at goal
- **Goal-driven**: assume goal is true → use rules to assert other facts as true → eventually arrive at known true facts

Applying Modus Ponens **backward**

Inference in Datalog (contd.)

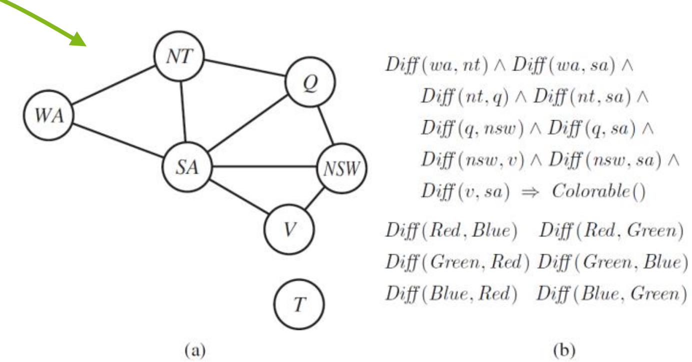
Forward chaining

- The data-driven approach:
  search for true antecedents («**if** clauses») → **infer** consequent («**then** clause») to be true → add this information to KB
- **Intuitive**ly understandable
- **Sound** and **complete** for Datalog
- Efficiently implementable for Datalog
  (a clause can be viewed as defining a CSP)
  → runs in **polynomial time**

- Humans control forward chaining carefully to not get flooded by irrelevant facts; but:
  “if I am indoors and I hear rain fall ➔ I might conclude that the picnic will be canceled”

Backward chaining

- The goal-driven approach: produces **no unnecessary facts**
- **Sound** and **complete** for Horn clauses
- Typically implemented using a form of SLD resolution (usually using **depth-first search**)
  → also used in pyDatalog



Map coloring as (a) a constraint graph and (b) as a single definite clause.

Datalog in practice

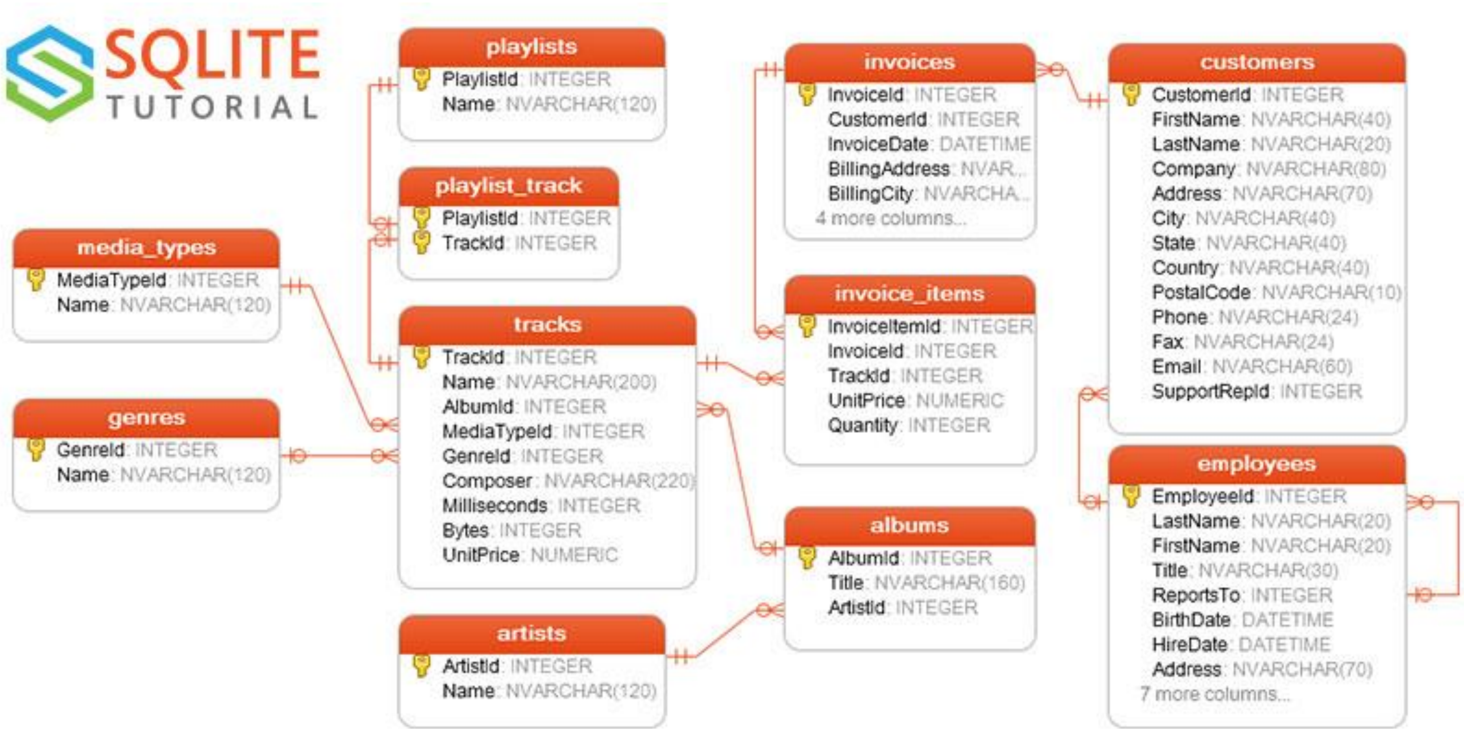
History and future

- Lots of **research** during the **1980's**, including many extensions
  → Ceri et al., *"What You Always Wanted to Know About Datalog (And Never Dared to Ask)"*,
  IEEE Trans. Knowledge & Data Engineering, 1989
- Ideas **influenced mainstream** database technology (e.g., recursion in SQL:1999) and the semantic web
  → https://en.wikipedia.org/wiki/Datalog
- **Resurge**d interest since the rise of **big data** (compare also B. Howe's lecture)
  → deMoor et al., "Datalog Reloaded – 1st International Workshop, Datalog 2010", Springer LNCS, 2010
- **LogicBlox** is a company build around Datalog (**product**, research and **tech. transfer**)
  → Aref, "Datalog for Enterprise Software – From Industrial Applications to Research", ICLP 2010

The pyDatalog interpreter

- Light weight, fast, and includes many extensions that facilitate efficiency and convenience
  - Memoization of intermediate results
  - Access to 11 SQL dialects via integration with SQLAlchemy
  - Includes aggregate functions and support for OOP
  - Easy mapping of logical terms to Python data structures or records from a DB
  - Not used often yet, but at least once in production

Example: Querying the chinook.db database



- A DB on music/media information and a company that sells them
- Access it from a terminal (DB in current directory, `sqlite` installed): `sqlite3 chinook.db`
- List all tables using the `sqlite` prompt: `.tables`

Example (contd.): DB query using pyDatalog

Accessing existing relations and fetching results

```
from sqlalchemy.ext.declarative import declarative_base; from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker; from pyDatalog import pyDatalog

# define a base class with SQLAlchemy and pyDatalog capabilities
Base = declarative_base(cls=pyDatalog.Mixin, metaclass=pyDatalog.sqlMetaMixin)

# load a database from the same directory and create a session, then associate it to the Base class
engine = create_engine('sqlite:///chinook.db') #, echo=False)
Session = sessionmaker(bind=engine)
session = Session()
Base.session = session

# classes that inherit from Base will now have both pyDatalog and SQLAlchemy capability
# the approach can be used to load an existing KB from a database relation, using __table_args__ :
class Track(Base):
    __tablename__ = 'tracks'
    __table_args__ = {'autoload':True, 'autoload_with':engine} #autoload the model

    def __repr__(self): #specifies how to print a Track
        return "'" + self.Name + "' costs $"  + str(self.UnitPrice)

# the Track class can now be used in in-line queries; example: which track is at least 5s long?
X = pyDatalog.Variable()
Track.Milliseconds[X] >= 5000000
print(X) #outputs ['Through a Looking Glass' costs $1.99, 'Occupation / Precipice' costs $1.99]
```

Installing SQLAlchemy: `conda install sqlalchemy`
Using SQLite: https://www.codeproject.com/Articles/850834/Installing-and-Using-SQLite-on-Windows
Finding the example DB: http://www.sqlitetutorial.net/sqlite-sample-database/ (→ see also DB schema on next slide)

Where's the intelligence?

Man vs. machine

Datalog makes the following **assumptions about the world**

- An agent's **knowledge** can be usefully described in terms of **individuals** and **relations** among individuals
- An agent's knowledge base consists of **definite and positive statements**
- The environment is **static**
- There are only **a finite number of individuals** of interest in the domain
- Each individual can be given a **unique name**

Under these assumptions, Datalog is a **powerful yet fast** system for **inference**

- **Modeling** the real world to conform to the assumptions is **up to the developer**

Review

- **Datalog** combines **expressive** power (about individuals and their relations) with **efficient** inference
- **Forward** and **backward chaining** are fast & complete for Horn clauses (Datalog)
- While **Datalog** might gain **popularity in big data** applications in the future, **logic** in general remains **very important for AI**

Zurich University
of Applied Sciences

APPENDIX

Propositional logic cheat sheet

$$
\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} & \rightarrow & \mathit{True} \mid \mathit{False} \mid P \mid Q \mid R \mid \ldots \\
\mathit{ComplexSentence} & \rightarrow & (\ \mathit{Sentence}\ ) \mid [\ \mathit{Sentence}\ ] \\
 & \mid & \neg\, \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \land \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \lor \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \;\Rightarrow\; \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \;\Leftrightarrow\; \mathit{Sentence} \\
\\
\text{OPERATOR PRECEDENCE} & : & \neg, \land, \lor, \Rightarrow, \Leftrightarrow
\end{array}
$$

Figure 7.7 A BNF (Backus–Naur Form) grammar of sentences in propositional logic, along with operator precedences, from highest to lowest.

P	Q	$\neg P$	$P \land Q$	$P \lor Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

Figure 7.8 Truth tables for the five logical connectives. To use the table to compute, for example, the value of $P \lor Q$ when P is true and Q is false, first look on the left for the row where P is *true* and Q is *false* (the third row). Then look in that row under the $P \lor Q$ column to see the result: *true*.

First order logic cheat sheet

Zurich University of Applied Sciences

$$
\begin{array}{rcl}
Sentence & \rightarrow & AtomicSentence \mid ComplexSentence \\
AtomicSentence & \rightarrow & Predicate \mid Predicate(Term, \ldots) \mid Term = Term \\
ComplexSentence & \rightarrow & (\ Sentence\ ) \mid [\ Sentence\ ] \\
& \mid & \neg\ Sentence \\
& \mid & Sentence \land Sentence \\
& \mid & Sentence \lor Sentence \\
& \mid & Sentence \ \Rightarrow\ Sentence \\
& \mid & Sentence \ \Leftrightarrow\ Sentence \\
& \mid & Quantifier\ Variable, \ldots\ Sentence \\
\\
Term & \rightarrow & Function(Term, \ldots) \\
& \mid & Constant \\
& \mid & Variable \\
\\
Quantifier & \rightarrow & \forall \mid \exists \\
Constant & \rightarrow & A \mid X_1 \mid John \mid \cdots \\
Variable & \rightarrow & a \mid x \mid s \mid \cdots \\
Predicate & \rightarrow & True \mid False \mid After \mid Loves \mid Raining \mid \cdots \\
Function & \rightarrow & Mother \mid LeftLeg \mid \cdots \\
\text{OPERATOR PRECEDENCE} & : & \neg, =, \land, \lor, \Rightarrow, \Leftrightarrow
\end{array}
$$

Figure 8.3 The syntax of first-order logic with equality, specified in Backus–Naur form (see page 1060 if you are not familiar with this notation). Operator precedences are specified, from highest to lowest. The precedence of quantifiers is such that a quantifier holds over everything to the right of it.

Datalog vs. SPARQL

Pro SPARQL

- Designed for graph queries

Con SPARQL

- Not algebraically closed (input is a graph, but output is a set of records)
- Limited expressiveness (no arbitrary recursion)
- If input is tabular, you have to shred it into a graph before using SPARQL
  (➔ often a 3x-x blow up in size!)
- Everything has to be in one graph