A rule-based recursive query language

```plaintext
father(alice, bob)
mother(alice, carla)
Parent(x, y) ← father(x, y)
Parent(x, y) ← mother(x, y)
SameGeneration(x, x)
SameGeneration(x, y) ← Parent(x, v) ∧ Parent(y, w) ∧ SameGeneration(v, w)
```

Perfect static optimisation for Datalog is undecidable

Datalog queries can be evaluated bottom-up or top-down

Simplest practical bottom-up technique: semi-naive evaluation
Semi-Naive Evaluation: Example

\[ e(1, 2) \quad e(2, 3) \quad e(3, 4) \quad e(4, 5) \]

\[(R1) \quad T(x, y) \leftarrow e(x, y)\]

\[(R2.1) \quad T(x, z) \leftarrow \Delta_i^T(x, y) \land T^i(y, z)\]

\[(R2.2') \quad T(x, z) \leftarrow T^{i-1}(x, y) \land \Delta_i^T(y, z)\]

How many body matches do we need to iterate over?

\[ T_P^0 = \emptyset \]  
initialisation

\[ T_P^1 = \{T(1, 2), T(2, 3), T(3, 4), T(4, 5)\} \quad 4 \times (R1) \]

\[ T_P^2 = T_P^1 \cup \{T(1, 3), T(2, 4), T(3, 5)\} \quad 3 \times (R2.1) \]

\[ T_P^3 = T_P^2 \cup \{T(1, 4), T(2, 5), T(1, 5)\} \quad 3 \times (R2.1), 2 \times (R2.2') \]

\[ T_P^4 = T_P^3 = T_P^\infty \quad 1 \times (R2.1), 1 \times (R2.2') \]

In total, we considered 14 matches to derive 11 facts
Semi-Naive Evaluation: Full Definition

In general, a rule of the form

\[ H(\vec{x}) \leftarrow e_1(\vec{y}_1) \land \ldots \land e_n(\vec{y}_n) \land I_1(\vec{z}_1) \land I_2(\vec{z}_2) \land \ldots \land I_m(\vec{z}_m) \]

is transformed into \( m \) rules

\[ H(\vec{x}) \leftarrow e_1(\vec{y}_1) \land \ldots \land e_n(\vec{y}_n) \land \Delta_i^1(\vec{z}_1) \land I_2(\vec{z}_2) \land \ldots \land I_m(\vec{z}_m) \]
\[ H(\vec{x}) \leftarrow e_1(\vec{y}_1) \land \ldots \land e_n(\vec{y}_n) \land I_1(\vec{z}_1) \land \Delta_i^2(\vec{z}_2) \land \ldots \land I_m(\vec{z}_m) \]
\[ \ldots \]
\[ H(\vec{x}) \leftarrow e_1(\vec{y}_1) \land \ldots \land e_n(\vec{y}_n) \land I_1(\vec{z}_1) \land I_2(\vec{z}_2) \land \ldots \land \Delta_i^m(\vec{z}_m) \]

Advantages and disadvantages:

- Huge improvement over naive evaluation
- Some redundant computations remain (see example)
- Some overhead for implementation (store level of entailments)
Top-Down Evaluation

**Idea:** we may not need to compute all derivations to answer a particular query

**Example 15.1:**

\[
\begin{align*}
    e(1, 2) & \quad e(2, 3) & \quad e(3, 4) & \quad e(4, 5) \\
    (R1) & \quad T(x, y) & \leftarrow & \quad e(x, y) \\
    (R2) & \quad T(x, z) & \leftarrow & \quad T(x, y) \land T(y, z) \\
    \text{Query}(z) & \leftarrow & \quad T(2, z)
\end{align*}
\]

The answers to Query are the T-successors of 2.

However, bottom-up computation would also produce facts like T(1, 4), which are neither directly nor indirectly relevant for computing the query result.
Assumption: For all techniques presented in this lecture, we assume that the given Datalog program is safe.

- This is without loss of generality (as shown in exercise).
- One can avoid this by adding more cases to algorithms.
Query-Subquery (QSQ)

QSQ is a technique for organising top-down Datalog query evaluation

**Main principles:**

- Apply **backward chaining/resolution**: start with query, find rules that can derive query, evaluate body atoms of those rules (subqueries) recursively
- Evaluate intermediate results “set-at-a-time” (using relational algebra on tables)
- Evaluate queries in a “data-driven” way, where operations are applied only to newly computed intermediate results (similar to idea in semi-naive evaluation)
- “Push” variable bindings (constants) from heads (queries) into bodies (subqueries)
- “Pass” variable bindings (constants) “sideways” from one body atom to the next

Details can be realised in several ways.
Adornments

To guide evaluation, we distinguish free and bound parameters in a predicate.

**Example 15.2:** If we want to derive atom $T(2, z)$ from the rule $T(x, z) \leftarrow T(x, y) \land T(y, z)$, then $x$ will be bound to 2, while $z$ is free.
To guide evaluation, we distinguish free and bound parameters in a predicate.

**Example 15.2:** If we want to derive atom $T(2, z)$ from the rule $T(x, z) \leftarrow T(x, y) \land T(y, z)$, then $x$ will be bound to 2, while $z$ is free.

We use adornments to denote the free/bound parameters in predicates.

**Example 15.3:**

\[
T_{bf}^{bf}(x, z) \leftarrow T_{bf}^{bf}(x, y) \land T_{bf}^{bf}(y, z)
\]

- since $x$ is bound in the head, it is also bound in the first atom
- any match for the first atom binds $y$, so $y$ is bound when evaluating the second atom (in left-to-right evaluation)
Adornments: Examples

The adornment of the head of a rule determines the adornments of the body atoms:

\[ R^{bbb}(x, y, z) \leftarrow R^{bbf}(x, y, v) \land R^{bbb}(x, v, z) \]
\[ R^{fbf}(x, y, z) \leftarrow R^{fbf}(x, y, v) \land R^{bbf}(x, v, z) \]
Adornments: Examples

The adornment of the head of a rule determines the adornments of the body atoms:

\[
\begin{align*}
R^{bb\!\!\!\!\!\!\!\!b\!\!\!\!\!\!\!\!b\!\!\!\!\!\!\!\!b}(x, y, z) & \leftarrow R^{bb\!\!\!\!\!\!\!\!f\!\!\!\!\!\!\!\!f\!\!\!\!\!\!\!\!b}(x, y, v) \land R^{bb\!\!\!\!\!\!\!\!b\!\!\!\!\!\!\!\!b\!\!\!\!\!\!\!\!b}(x, v, z) \\
R^{ff\!\!\!\!\!\!\!\!b\!\!\!\!\!\!\!\!b}(x, y, z) & \leftarrow R^{ff\!\!\!\!\!\!\!\!f\!\!\!\!\!\!\!\!b\!\!\!\!\!\!\!\!b}(x, y, v) \land R^{ff\!\!\!\!\!\!\!\!b\!\!\!\!\!\!\!\!f\!\!\!\!\!\!\!\!b}(x, v, z)
\end{align*}
\]

The order of body predicates affects the adornment:

\[
\begin{align*}
S^{ff\!\!\!\!\!\!\!\!f\!\!\!\!\!\!\!\!f\!\!\!\!\!\!\!\!f}(x, y, z) & \leftarrow T^{ff\!\!\!\!\!\!\!\!f\!\!\!\!\!\!\!\!f}(x, v) \land T^{ff\!\!\!\!\!\!\!\!f\!\!\!\!\!\!\!\!f}(y, w) \land R^{bb\!\!\!\!\!\!\!\!f\!\!\!\!\!\!\!\!f\!\!\!\!\!\!\!\!b\!\!\!\!\!\!\!\!f\!\!\!\!\!\!\!\!b}(v, w, z) \\
S^{ff\!\!\!\!\!\!\!\!f\!\!\!\!\!\!\!\!f}(x, y, z) & \leftarrow R^{ff\!\!\!\!\!\!\!\!f\!\!\!\!\!\!\!\!f\!\!\!\!\!\!\!\!f\!\!\!\!\!\!\!\!f\!\!\!\!\!\!\!\!f}(v, w, z) \land T^{ff\!\!\!\!\!\!\!\!f\!\!\!\!\!\!\!\!f\!\!\!\!\!\!\!\!f}(x, v) \land T^{ff\!\!\!\!\!\!\!\!f\!\!\!\!\!\!\!\!f}(y, w)
\end{align*}
\]

\[\leadsto\] For optimisation, some orders might be better than others
Auxiliary Relations for QSQ

To control evaluation, we store intermediate results in auxiliary relations.

When we “call” a rule with a head where some variables are bound, we need to provide the bindings as input

\[ \sim \text{ for adorned relation } R^\alpha, \text{ we use an auxiliary relation } \text{input}_R^\alpha \]

\[ \sim \text{ arity of } \text{input}_R^\alpha = \text{ number of } b \text{ in } \alpha \]

The result of calling a rule should be the “completed” input, with values for the unbound variables added

\[ \sim \text{ for adorned relation } R^\alpha, \text{ we use an auxiliary relation } \text{output}_R^\alpha \]

\[ \sim \text{ arity of } \text{output}_R^\alpha = \text{ arity of } R (= \text{ length of } \alpha) \]
Auxiliary Relations for QSQ (2)

When evaluating body atoms from left to right, we use supplementary relations $\text{sup}_i$

$\leadsto$ bindings required to evaluate rest of rule after the $i$th body atom

$\leadsto$ the first set of bindings $\text{sup}_0$ comes from input $\alpha_R$

$\leadsto$ the last set of bindings $\text{sup}_n$ go to output $\alpha_R$

Example 15.4:

$T_{bf}(x, z) \leftarrow T_{bf}(x, y) \land T_{bf}(y, z)$

$\uparrow_u \uparrow_u$

input $bf_T \Rightarrow \text{sup}_0[x]

\text{sup}_1[x, y]

\text{sup}_2[x, z] \Rightarrow \text{output} \quad (\text{we use “named” notation like } [x, y]\text{ to suggest what to join on; the relations are the same})
Auxiliary Relations for QSQ (2)

When evaluating body atoms from left to right, we use supplementary relations $\text{sup}_i$

$\leadsto$ bindings required to evaluate rest of rule after the $i$th body atom

$\leadsto$ the first set of bindings $\text{sup}_0$ comes from input $\text{bf}_R^\alpha$

$\leadsto$ the last set of bindings $\text{sup}_n$ go to output $\text{bf}_R^\alpha$

Example 15.4:

\[
\text{bf}_T^f(x, z) \leftarrow \text{bf}_T^f(x, y) \land \text{bf}_T^f(y, z)
\]

\[
\uparrow \quad \Downarrow \uparrow \quad \Downarrow \quad \Downarrow
\]

input $\text{bf}_T^f \Rightarrow \text{sup}_0[x] \quad \text{sup}_1[x, y] \quad \text{sup}_2[x, z] \Rightarrow \text{output}_T^f
\]

- $\text{sup}_0[x]$ is copied from input $\text{bf}_T^f[x]$ (with some exceptions, see exercise)
- $\text{sup}_1[x, y]$ is obtained by joining tables $\text{sup}_0[x]$ and output $\text{bf}_T^f[x, y]$
- $\text{sup}_2[x, z]$ is obtained by joining tables $\text{sup}_1[x, y]$ and output $\text{bf}_T^f[y, z]$
- $\text{output}_T^f[x, z]$ is copied from $\text{sup}_2[x, z]$

(we use “named” notation like $[x, y]$ to suggest what to join on; the relations are the same)
The set of all auxiliary relations is called a **QSQ template** (for the given set of adorned rules)

**General evaluation:**

- add new tuples to auxiliary relations until reaching a fixed point
- evaluation of a rule can proceed as sketched on previous slide
- in addition, whenever new tuples are added to a sup relation that feeds into an IDB atom, the input relation of this atom is extended to include all binding given by sup (may trigger subquery evaluation)

→ there are many strategies for implementing this general scheme
QSQ Evaluation

The set of all auxiliary relations is called a **QSQ template** (for the given set of adorned rules)

**General evaluation:**

- add new tuples to auxiliary relations until reaching a fixed point
- evaluation of a rule can proceed as sketched on previous slide
- in addition, whenever new tuples are added to a sup relation that feeds into an IDB atom, the input relation of this atom is extended to include all binding given by sup (may trigger subquery evaluation)

→ there are many strategies for implementing this general scheme

**Notation:**

- for an EDB atom $A$, we write $A^I$ for table that consists of all matches for $A$ in the database
Recursive QSQ (QSQR) takes a “depth-first” approach to QSQ

**Evaluation of single rule in QSQR:**

Given: adorned rule \( r \) with head predicate \( R^\alpha \); current values of all QSQ relations

1. Copy tuples \( \text{input}^\alpha_R \) (that unify with rule head) to \( \text{sup}^r_0 \)

2. For each body atom \( A_1, \ldots, A_n \), do:
   - If \( A_i \) is an EDB atom, compute \( \text{sup}^r_i \) as projection of \( \text{sup}^r_{i-1} \bowtie A_i^T \)
   - If \( A_i \) is an IDB atom with adorned predicate \( S^\beta \):
     - (a) Add new bindings from \( \text{sup}^r_{i-1} \), combined with constants in \( A_i \), to \( \text{input}^\beta_S \)
     - (b) If \( \text{input}^\beta_S \) changed, recursively evaluate all rules with head predicate \( S^\beta \)
     - (c) Compute \( \text{sup}^r_i \) as projection of \( \text{sup}^r_{i-1} \bowtie \text{output}^\beta_S \)

3. Add tuples in \( \text{sup}^r_n \) to \( \text{output}^\alpha_R \)
Evaluation of query in QSQR:

Given: a Datalog program $P$ and a conjunctive query $q[\vec{x}]$ (possibly with constants)

(1) Create an adorned program $P^a$:
   - Turn the query $q[\vec{x}]$ into an adorned rule $\text{Query}^{f\cdots f}(\vec{x}) \leftarrow q[\vec{x}]$
   - Recursively create adorned rules from rules in $P$ for all adorned predicates in $P^a$.

(2) Initialise all auxiliary relations to empty sets.

(3) Evaluate the rule $\text{Query}^{f\cdots f}(\vec{x}) \leftarrow q[\vec{x}]$.

   Repeat until no new tuples are added to any QSQ relation.

(4) Return output $\text{Query}$. 
QSQR Transformation: Example

Predicates S (same generation), p (parent), h (human)

\[
\begin{align*}
S(x, x) & \leftarrow h(x) \\
S(x, y) & \leftarrow p(x, w) \land S(v, w) \land p(y, v)
\end{align*}
\]

with query S(1, x).
QSQR Transformation: Example

Predicates S (same generation), p (parent), h (human)

\[
S(x, x) \leftarrow h(x) \\
S(x, y) \leftarrow p(x, w) \land S(v, w) \land p(y, v)
\]

with query \(S(1, x)\).

\[\sim\] Query rule: \(\text{Query}(x) \leftarrow S(1, x)\)

Transformed rules:
QSQR Transformation: Example

Predicates S (same generation), p (parent), h (human)

\[
\begin{align*}
S(x, x) & \leftarrow h(x) \\
S(x, y) & \leftarrow p(x, w) \land S(v, w) \land p(y, v)
\end{align*}
\]

with query \( S(1, x) \).

\[ \leadsto \text{Query rule: } \text{Query}(x) \leftarrow S(1, x) \]

Transformed rules:

\[
\begin{align*}
\text{Query}^f(x) & \leftarrow S^{bf}(1, x)
\end{align*}
\]
QSQR Transformation: Example

Predicates S (same generation), p (parent), h (human)

\[
\begin{align*}
S(x, x) & \leftarrow h(x) \\
S(x, y) & \leftarrow p(x, w) \land S(v, w) \land p(y, v)
\end{align*}
\]

with query \( S(1, x) \).

\( \sim \) Query rule: \( \text{Query}(x) \leftarrow S(1, x) \)

Transformed rules:

\[
\begin{align*}
\text{Query}^f(x) & \leftarrow S^{bf}(1, x) \\
S^{bf}(x, x) & \leftarrow h(x)
\end{align*}
\]
QSQR Transformation: Example

Predicates \( S \) (same generation), \( p \) (parent), \( h \) (human)

\[
\begin{align*}
S(x, x) & \leftarrow h(x) \\
S(x, y) & \leftarrow p(x, w) \land S(v, w) \land p(y, v)
\end{align*}
\]

with query \( S(1, x) \).
\sim Query rule: Query(x) \leftarrow S(1, x)

Transformed rules:

\[
\begin{align*}
\text{Query}^f(x) & \leftarrow S^{bf}(1, x) \\
S^{bf}(x, x) & \leftarrow h(x) \\
S^{bf}(x, y) & \leftarrow p(x, w) \land S^{fb}(v, w) \land p(y, v)
\end{align*}
\]
QSQR Transformation: Example

Predicates S (same generation), p (parent), h (human)

\[
\begin{align*}
S(x, x) & \leftarrow h(x) \\
S(x, y) & \leftarrow p(x, w) \land S(v, w) \land p(y, v)
\end{align*}
\]

with query \(S(1, x)\).

\(~\) Query rule: \(\text{Query}(x) \leftarrow S(1, x)\)

Transformed rules:

\[
\begin{align*}
\text{Query}^f(x) & \leftarrow S^{bf}(1, x) \\
S^{bf}(x, x) & \leftarrow h(x) \\
S^{bf}(x, y) & \leftarrow p(x, w) \land S^{fb}(v, w) \land p(y, v) \\
S^{fb}(x, x) & \leftarrow h(x)
\end{align*}
\]
QSQR Transformation: Example

Predicates S (same generation), p (parent), h (human)

\[
\begin{align*}
\text{S}(x, x) & \leftarrow h(x) \\
\text{S}(x, y) & \leftarrow p(x, w) \land \text{S}(v, w) \land p(y, v)
\end{align*}
\]

with query \(\text{S}(1, x)\).

\(\sim\) Query rule: Query\((x) \leftarrow \text{S}(1, x)\)

Transformed rules:

\[
\begin{align*}
\text{Query}^f(x) & \leftarrow \text{S}^{bf}(1, x) \\
\text{S}^{bf}(x, x) & \leftarrow h(x) \\
\text{S}^{bf}(x, y) & \leftarrow p(x, w) \land \text{S}^{fb}(v, w) \land p(y, v) \\
\text{S}^{fb}(x, x) & \leftarrow h(x) \\
\text{S}^{fb}(x, y) & \leftarrow p(x, w) \land \text{S}^{fb}(v, w) \land p(y, v)
\end{align*}
\]
Magic Sets

QSQ(R) is a goal directed procedure: it tries to derive results for a specific query.

Semi-naive evaluation is not goal directed: it computes all entailed facts.

Can a bottom-up technique be goal-directed?
Magic Sets

QSQ(R) is a goal directed procedure: it tries to derive results for a specific query.

Semi-naive evaluation is not goal directed: it computes all entailed facts.

Can a bottom-up technique be goal-directed?
\[ \sim \text{yes, by magic} \]

Magic Sets

- “Simulation” of QSQ by Datalog rules
- Can be evaluated bottom up, e.g., with semi-naive evaluation
- The “magic sets” are the sets of tuples stored in the auxiliary relations
- Several other variants of the method exist
Magic Sets as Simulation of QSQ

Idea: the information flow in QSQ(R) mainly uses join and projection

~ can we just implement this in Datalog?
**Magic Sets as Simulation of QSQ**

**Idea:** the information flow in QSQ(R) mainly uses join and projection

∼ can we just implement this in Datalog?

**Example 15.5:** The QSQ information flow

\[ T^{bf}(x, z) \leftarrow T^{bf}(x, y) \land T^{bf}(y, z) \]

\[ \uparrow \quad \downarrow \quad \uparrow \quad \downarrow \]

input\(^{bf}_T \Rightarrow sup_0[x] \quad sup_1[x, y] \quad sup_2[x, z] \Rightarrow output\(^{bf}_T \]

could be expressed using rules:

\[ sup_0(x) \leftarrow input\(^{bf}_T (x) \]

\[ sup_1(x, y) \leftarrow sup_0(x) \land output\(^{bf}_T (x, y) \]

\[ sup_2(x, z) \leftarrow sup_1(x, y) \land output\(^{bf}_T (y, z) \]

\[ output\(^{bf}_T (x, z) \leftarrow sup_2(x, z) \]
Observation: \( \text{sup}_0(x) \) and \( \text{sup}_2(x, z) \) are redundant. Simpler:

\[
\begin{align*}
\text{sup}_1(x, y) & \leftarrow \text{input}^T_{bf}(x) \land \text{output}^T_{bf}(x, y) \\
\text{output}^T_{bf}(x, z) & \leftarrow \text{sup}_1(x, y) \land \text{output}^T_{bf}(y, z)
\end{align*}
\]
Observation: $\sup_0(x)$ and $\sup_2(x, z)$ are redundant. Simpler:

\[
\begin{align*}
\sup_1(x, y) & \leftarrow \text{input}^\text{bf}_T(x) \land \text{output}^\text{bf}_T(x, y) \\
\text{output}^\text{bf}_T(x, z) & \leftarrow \sup_1(x, y) \land \text{output}^\text{bf}_T(y, z)
\end{align*}
\]

We still need to “call” subqueries recursively:

\[
\text{input}^\text{bf}_T(y) \leftarrow \sup_1(x, y)
\]

It is easy to see how to do this for arbitrary adorned rules.
A Note on Constants

Constants in rule bodies must lead to bindings in the subquery.
A Note on Constants

Constants in rule bodies must lead to bindings in the subquery.

Example 15.6: The following rule is correctly adorned

\[ R^{bf}(x, y) \leftarrow T^{bbf}(x, a, y) \]

This leads to the following rules using Magic Sets:

\[
\begin{align*}
\text{output}^{bf}_R(x, y) & \leftarrow \text{input}^{bf}_R(x) \land \text{output}^{bbf}_T(x, a, y) \\
\text{input}^{bbf}_T(x, a) & \leftarrow \text{input}^{bf}_R(x)
\end{align*}
\]

Note that we do not need to use auxiliary predicates sup\(_0\) or sup\(_1\) here, by the simplification on the previous slide.
Magic Sets: Summary

A goal-directed bottom-up technique:

- Rewritten program rules can be constructed on the fly
- Bottom-up evaluation can be semi-naive (avoid repeated rule applications)
- Supplementary relations can be cached in between queries
Magic Sets: Summary

A goal-directed bottom-up technique:

- Rewritten program rules can be constructed on the fly
- Bottom-up evaluation can be semi-naive (avoid repeated rule applications)
- Supplementary relations can be cached in between queries

Nevertheless, a full materialisation might be better, if

- Database does not change very often (materialisation as one-time investment)
- Queries are very diverse and may use any IDB relation (bad for caching supplementary relations)

\[\rightarrow\] semi-naive evaluation is still very common in practice
Implementation
How to Implement Datalog

We saw several evaluation methods:

- Semi-naive evaluation
- QSQ(R)
- Magic Sets

Don’t we have enough algorithms by now?
How to Implement Datalog

We saw several evaluation methods:

- Semi-naive evaluation
- QSQR
- Magic Sets

Don’t we have enough algorithms by now?

No. In fact, we are still far from actual algorithms.

Issues on the way from “evaluation method” to basic algorithm:

- Data structures! (Especially: how to store derivations?)
- Joins! (low-level algorithms; optimisations)
- Duplicate elimination! (major performance factor)
- Optimisations! (further ideas for reducing redundancy)
- Parallelism! (using multiple CPUs)
- ...
General concerns

System implementations need to decide on their mode of operation:

- Interactive service vs. batch process
- Scale? (related: what kind of memory and compute infrastructure to target?)
- Computing the complete least model vs. answering specific queries
- Static vs. dynamic inputs (will data change? will rules change?)
- Which data sources should be supported?
- Should results be cached? Do we to update caches (view maintenance)?
- Is intra-query parallelism desirable? On which level and for how many CPUs?
- …
Datalog as a Special Case

Datalog is a special case of many approaches, leading to very diverse implementation techniques.

- Prolog is essentially “Datalog with function symbols” (and many built-ins).
- Answer Set Programming is “Datalog extended with non-monotonic negation and disjunction”.
- Production Rules use “bottom-up rule reasoning with operational, non-monotonic built-ins”.
- Recursive SQL Queries are a syntactically restricted set of Datalog rules.

Different scenarios, different optimal solutions.

Not all implementations are complete (e.g., Prolog).

Markus Krötzsch, 5th June 2019
Datalog as a Special Case

Datalog is a special case of many approaches, leading to very diverse implementation techniques.

- **Prolog** is essentially “Datalog with function symbols” (and many built-ins).
Datalog as a Special Case

Datalog is a special case of many approaches, leading to very diverse implementation techniques.

- **Prolog** is essentially “Datalog with function symbols” (and many built-ins).
- **Answer Set Programming** is “Datalog extended with non-monotonic negation and disjunction”
Datalog as a Special Case

Datalog is a special case of many approaches, leading to very diverse implementation techniques.

- **Prolog** is essentially “Datalog with function symbols” (and many built-ins).
- **Answer Set Programming** is “Datalog extended with non-monotonic negation and disjunction”
- **Production Rules** use “bottom-up rule reasoning with operational, non-monotonic built-ins”
Datalog as a Special Case

Datalog is a special case of many approaches, leading to very diverse implementation techniques.

- **Prolog** is essentially “Datalog with function symbols” (and many built-ins).
- **Answer Set Programming** is “Datalog extended with non-monotonic negation and disjunction”
- **Production Rules** use “bottom-up rule reasoning with operational, non-monotonic built-ins”
- **Recursive SQL Queries** are a syntactically restricted set of Datalog rules
Datalog as a Special Case

Datalog is a special case of many approaches, leading to very diverse implementation techniques.

- **Prolog** is essentially “Datalog with function symbols” (and many built-ins).
- **Answer Set Programming** is “Datalog extended with non-monotonic negation and disjunction”
- **Production Rules** use “bottom-up rule reasoning with operational, non-monotonic built-ins”
- **Recursive SQL Queries** are a syntactically restricted set of Datalog rules

\[ \leadsto \text{Different scenarios, different optimal solutions} \]
\[ \leadsto \text{Not all implementations are complete (e.g., Prolog)} \]
Datalog Implementation in Practice

Dedicated Datalog engines as of 2018 (incomplete):

- **RDFox**  Fast in-memory RDF database with runtime materialisation and updates
- **VLog**  Fast in-memory Datalog materialisation with bindings to several databases, including RDF and RDBMS (co-developed at TU Dresden)
- **Llunatic**  PostgreSQL-based implementation of a rule engine
- **Graal**  In-memory rule engine with RDBMS bindings
- **SociaLite** and **EmptyHeaded**  Datalog-based languages and engines for social network analysis
- **DeepDive**  Data analysis platform with support for Datalog-based language “DDlog”
- **LogicBlox**  Big data analytics platform that uses Datalog rules (commercial, discontinued?)
- **DLV**  Answer set programming engine that is usable on Datalog programs (commercial)
- **Datatomic**  Distributed, versioned database using Datalog as main query language (commercial)
- **E**  Fast theorem prover for first-order logic with equality; can be used on Datalog as well
- …

∼ Extremely diverse tools for very different requirements
Summary and Outlook

Several implementation techniques for Datalog

- bottom up (from the data) or top down (from the query)
- goal-directed (for a query) or not

Top-down: Query-Subquery (QSQ) approach (goal-directed)

Bottom-up:

- naive evaluation (not goal-directed)
- semi-naive evaluation (not goal-directed)
- Magic Sets (goal-directed)

Next topics:

- Graph databases and path queries
- Dependencies