TECHNISCHE UNIVERSITÄT DRESDEN

DATABASE THEORY

Lecture 15: Datalog Evaluation (2)

Markus Krötzsch

Knowledge-Based Systems

TU Dresden, 26th June 2018

Review: Datalog Evaluation

A rule-based recursive query language

$$
\begin{aligned}
\text{father}(\text{alice}, \text{bob}) \\
\text{mother}(\text{alice}, \text{carla}) \\
\text{Parent}(x, y) &\leftarrow \text{father}(x, y) \\
\text{Parent}(x, y) &\leftarrow \text{mother}(x, y) \\
\text{SameGeneration}(x, x) \\
\text{SameGeneration}(x, y) &\leftarrow \text{Parent}(x, v) \wedge \text{Parent}(y, w) \wedge \text{SameGeneration}(v, w)
\end{aligned}
$$

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

$$\mathsf{e}(1,2) \quad \mathsf{e}(2,3) \quad \mathsf{e}(3,4) \quad \mathsf{e}(4,5)$$

$$(R1) \qquad \mathsf{T}(x,y) \leftarrow \mathsf{e}(x,y)$$

$$(R2.1) \qquad \mathsf{T}(x,z) \leftarrow \Delta^i_{\mathsf{T}}(x,y) \wedge \mathsf{T}^i(y,z)$$

$$(R2.2') \qquad \mathsf{T}(x,z) \leftarrow \mathsf{T}^{i-1}(x,y) \wedge \Delta^i_{\mathsf{T}}(y,z)$$

How many body matches do we need to iterate over?

$T_P^0 = \emptyset$	initialisation
$T_P^1 = \{\mathsf{T}(1,2), \mathsf{T}(2,3), \mathsf{T}(3,4), \mathsf{T}(4,5)\}$	$4 \times (R1)$
$T_P^2 = T_P^1 \cup \{\mathsf{T}(1,3), \mathsf{T}(2,4), \mathsf{T}(3,5)\}$	$3 \times (R2.1)$
$T_P^3 = T_P^2 \cup \{\mathsf{T}(1,4), \mathsf{T}(2,5), \mathsf{T}(1,5)\}$	$3 \times (R2.1), 2 \times (R2.2')$
$T_P^4 = T_P^3 = T_P^\infty$	$1 \times (R2.1), 1 \times (R2.2')$

In total, we considered 14 matches to derive 11 facts

Semi-Naive Evaluation: Full Definition

In general, a rule of the form

$$\mathsf{H}(\vec{x}) \leftarrow \mathsf{e}_1(\vec{y}_1) \wedge \ldots \wedge \mathsf{e}_n(\vec{y}_n) \wedge \mathsf{I}_1(\vec{z}_1) \wedge \mathsf{I}_2(\vec{z}_2) \wedge \ldots \wedge \mathsf{I}_m(\vec{z}_m)$$

is transformed into m rules

$$\mathsf{H}(\vec{x}) \leftarrow \mathsf{e}_1(\vec{y}_1) \wedge \ldots \wedge \mathsf{e}_n(\vec{y}_n) \wedge \Delta^i_{\mathsf{I}_1}(\vec{z}_1) \wedge \mathsf{I}^i_2(\vec{z}_2) \wedge \ldots \wedge \mathsf{I}^i_m(\vec{z}_m)$$

$$\mathsf{H}(\vec{x}) \leftarrow \mathsf{e}_1(\vec{y}_1) \wedge \ldots \wedge \mathsf{e}_n(\vec{y}_n) \wedge \mathsf{I}^{i-1}_1(\vec{z}_1) \wedge \Delta^i_{\mathsf{I}_2}(\vec{z}_2) \wedge \ldots \wedge \mathsf{I}^i_m(\vec{z}_m)$$

$$\ldots$$

$$\mathsf{H}(\vec{x}) \leftarrow \mathsf{e}_1(\vec{y}_1) \wedge \ldots \wedge \mathsf{e}_n(\vec{y}_n) \wedge \mathsf{I}^{i-1}_1(\vec{z}_1) \wedge \mathsf{I}^{i-1}_2(\vec{z}_2) \wedge \ldots \wedge \Delta^i_{\mathsf{I}_m}(\vec{z}_m)$$

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:

$$\mathsf{e}(1,2) \quad \mathsf{e}(2,3) \quad \mathsf{e}(3,4) \quad \mathsf{e}(4,5)$$

$$(R1) \qquad \mathsf{T}(x,y) \leftarrow \mathsf{e}(x,y)$$

$$(R2) \qquad \mathsf{T}(x,z) \leftarrow \mathsf{T}(x,y) \wedge \mathsf{T}(y,z)$$

$$\mathsf{Query}(z) \leftarrow \mathsf{T}(2,z)$$

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

However, bottom-up computation would also produce facts like $\mathsf{T}(1,4)$, which are neither directly nor indirectly relevant for computing the query result.

Assumption

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 $\mathsf{T}(2, z)$ from the rule $\mathsf{T}(x, z) \leftarrow \mathsf{T}(x, y) \wedge \mathsf{T}(y, z)$, then x will be bound to 2, while z is free.

Adornments

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

Example 15.2: If we want to derive atom $\mathsf{T}(2, z)$ from the rule $\mathsf{T}(x, z) \leftarrow \mathsf{T}(x, y) \wedge \mathsf{T}(y, z)$, 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:

$$\mathsf{T}^{bf}(x, z) \leftarrow \mathsf{T}^{bf}(x, y) \wedge \mathsf{T}^{bf}(y, z)$$

- 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:

$$\mathsf{R}^{bbb}(x, y, z) \leftarrow \mathsf{R}^{bbf}(x, y, v) \wedge \mathsf{R}^{bbb}(x, v, z)$$

$$\mathsf{R}^{fbf}(x, y, z) \leftarrow \mathsf{R}^{fbf}(x, y, v) \wedge \mathsf{R}^{bbf}(x, v, z)$$

Adornments: Examples

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

$$\mathsf{R}^{bbb}(x, y, z) \leftarrow \mathsf{R}^{bbf}(x, y, v) \wedge \mathsf{R}^{bbb}(x, v, z)$$
$$\mathsf{R}^{fbf}(x, y, z) \leftarrow \mathsf{R}^{fbf}(x, y, v) \wedge \mathsf{R}^{bbf}(x, v, z)$$

The order of body predicates affects the adornment:

$$\mathsf{S}^{fff}(x, y, z) \leftarrow \mathsf{T}^{ff}(x, v) \wedge \mathsf{T}^{ff}(y, w) \wedge \mathsf{R}^{bbf}(v, w, z)$$
$$\mathsf{S}^{fff}(x, y, z) \leftarrow \mathsf{R}^{fff}(v, w, z) \wedge \mathsf{T}^{fb}(x, v) \wedge \mathsf{T}^{fb}(y, w)$$

$\leadsto$ 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

$\leadsto$ for adorned relation R^α, we use an auxiliary relation $\mathsf{input}_\mathsf{R}^\alpha$

$\leadsto$ arity of $\mathsf{input}_\mathsf{R}^\alpha$ = number of b in α

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

$\leadsto$ for adorned relation R^α, we use an auxiliary relation $\mathsf{output}_\mathsf{R}^\alpha$

$\leadsto$ arity of $\mathsf{output}_\mathsf{R}^\alpha$ = arity of R (= length of α)

Auxiliary Relations for QSQ (2)

When evaluating body atoms from left to right, we use supplementary relations sup_i

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

$\leadsto$ the first set of bindings sup_0 comes from $\mathsf{input}^{\alpha}_{\mathsf{R}}$

$\leadsto$ the last set of bindings sup_n go to $\mathsf{output}^{\alpha}_{\mathsf{R}}$

Auxiliary Relations for QSQ (2)

When evaluating body atoms from left to right, we use supplementary relations sup_i

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

$\leadsto$ the first set of bindings sup_0 comes from $\mathsf{input}^{\alpha}_{\mathsf{R}}$

$\leadsto$ the last set of bindings sup_n go to $\mathsf{output}^{\alpha}_{\mathsf{R}}$

Example 15.4:

$$\begin{array}{ccccccc} & & \mathsf{T}^{bf}(x,z) \leftarrow \mathsf{T}^{bf}(x,y) & \wedge & \mathsf{T}^{bf}(y,z) & & \\ & & \Uparrow & \searrow\ \Uparrow & & \searrow & \\ \mathsf{input}^{bf}_{\mathsf{T}} & \Rightarrow & \mathsf{sup}_0[x] & \mathsf{sup}_1[x,y] & & \mathsf{sup}_2[x,z] & \Rightarrow \mathsf{output}^{bf}_{\mathsf{T}} \end{array}$$

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

(we use "named" notation like $[x,y]$ to suggest what to join on; the relations are the same)

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)

$\leadsto$ 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)

$\leadsto$ there are many strategies for implementing this general scheme

Notation:

- for an EDB atom A, we write $A^{\mathcal{I}}$ for table that consists of all matches for A in the database

Recursive QSQ

Recursive QSQ (QSQR) takes a "depth-first" approach to QSQ

Evaluation of single rule in QSQR:

Given: adorned rule r with head predicate R^{α}; current values of all QSQ relations

(1) Copy tuples $\mathsf{input}^{\alpha}_{\mathsf{R}}$ (that unify with rule head) to sup^{r}_{0}

(2) For each body atom $A_1, \ldots, A_n$, do:

- If A_i is an EDB atom, compute sup^{r}_{i} as projection of $\mathsf{sup}^{r}_{i-1} \bowtie A_i^{\mathcal{I}}$
- If A_i is an IDB atom with adorned predicate S^{β}:
  (a) Add new bindings from sup^{r}_{i-1}, combined with constants in A_i, to $\mathsf{input}^{\beta}_{\mathsf{S}}$
  (b) If $\mathsf{input}^{\beta}_{\mathsf{S}}$ changed, recursively evaluate all rules with head predicate S^{β}
  (c) Compute sup^{r}_{i} as projection of $\mathsf{sup}^{r}_{i-1} \bowtie \mathsf{output}^{\beta}_{\mathsf{S}}$

(3) Add tuples in sup^{r}_{n} to $\mathsf{output}^{\alpha}_{\mathsf{R}}$

QSQR Algorithm

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 $\text{Query}^{ff\ldots f}(\vec{x}) \leftarrow q[\vec{x}]$
   - 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 $\text{Query}^{ff\ldots f}(\vec{x}) \leftarrow q[\vec{x}]$.
Repeat until no new tuples are added to any QSQ relation.

(4) Return $\text{output}^{ff\ldots f}_{\text{Query}}$.

QSQR Transformation: Example

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

$$\mathsf{S}(x,x) \leftarrow \mathsf{h}(x)$$

$$\mathsf{S}(x,y) \leftarrow \mathsf{p}(x,w) \wedge \mathsf{S}(v,w) \wedge \mathsf{p}(y,v)$$

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

QSQR Transformation: Example

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

$$\mathsf{S}(x,x) \leftarrow \mathsf{h}(x)$$
$$\mathsf{S}(x,y) \leftarrow \mathsf{p}(x,w) \wedge \mathsf{S}(v,w) \wedge \mathsf{p}(y,v)$$

with query $\mathsf{S}(1,x)$.
$\leadsto$ Query rule: $\mathsf{Query}(x) \leftarrow \mathsf{S}(1,x)$

Transformed rules:

QSQR Transformation: Example

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

$$\mathsf{S}(x,x) \leftarrow \mathsf{h}(x)$$

$$\mathsf{S}(x,y) \leftarrow \mathsf{p}(x,w) \wedge \mathsf{S}(v,w) \wedge \mathsf{p}(y,v)$$

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

$\leadsto$ Query rule: $\mathsf{Query}(x) \leftarrow \mathsf{S}(1,x)$

Transformed rules:

$$\mathsf{Query}^{f}(x) \leftarrow \mathsf{S}^{bf}(1,x)$$

QSQR Transformation: Example

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

$$\mathsf{S}(x, x) \leftarrow \mathsf{h}(x)$$

$$\mathsf{S}(x, y) \leftarrow \mathsf{p}(x, w) \wedge \mathsf{S}(v, w) \wedge \mathsf{p}(y, v)$$

with query $\mathsf{S}(1, x)$.
$\leadsto$ Query rule: $\mathsf{Query}(x) \leftarrow \mathsf{S}(1, x)$

Transformed rules:

$$\mathsf{Query}^{f}(x) \leftarrow \mathsf{S}^{bf}(1, x)$$

$$\mathsf{S}^{bf}(x, x) \leftarrow \mathsf{h}(x)$$

QSQR Transformation: Example

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

$$\mathsf{S}(x,x) \leftarrow \mathsf{h}(x)$$
$$\mathsf{S}(x,y) \leftarrow \mathsf{p}(x,w) \wedge \mathsf{S}(v,w) \wedge \mathsf{p}(y,v)$$

with query $\mathsf{S}(1,x)$.
$\leadsto$ Query rule: $\mathsf{Query}(x) \leftarrow \mathsf{S}(1,x)$

Transformed rules:

$$\mathsf{Query}^{f}(x) \leftarrow \mathsf{S}^{bf}(1,x)$$
$$\mathsf{S}^{bf}(x,x) \leftarrow \mathsf{h}(x)$$
$$\mathsf{S}^{bf}(x,y) \leftarrow \mathsf{p}(x,w) \wedge \mathsf{S}^{fb}(v,w) \wedge \mathsf{p}(y,v)$$

QSQR Transformation: Example

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

$$\mathsf{S}(x,x) \leftarrow \mathsf{h}(x)$$
$$\mathsf{S}(x,y) \leftarrow \mathsf{p}(x,w) \wedge \mathsf{S}(v,w) \wedge \mathsf{p}(y,v)$$

with query $\mathsf{S}(1,x)$.
$\leadsto$ Query rule: $\mathsf{Query}(x) \leftarrow \mathsf{S}(1,x)$

Transformed rules:

$$\mathsf{Query}^{f}(x) \leftarrow \mathsf{S}^{bf}(1,x)$$
$$\mathsf{S}^{bf}(x,x) \leftarrow \mathsf{h}(x)$$
$$\mathsf{S}^{bf}(x,y) \leftarrow \mathsf{p}(x,w) \wedge \mathsf{S}^{fb}(v,w) \wedge \mathsf{p}(y,v)$$
$$\mathsf{S}^{fb}(x,x) \leftarrow \mathsf{h}(x)$$

QSQR Transformation: Example

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

$$\mathsf{S}(x,x) \leftarrow \mathsf{h}(x)$$
$$\mathsf{S}(x,y) \leftarrow \mathsf{p}(x,w) \wedge \mathsf{S}(v,w) \wedge \mathsf{p}(y,v)$$

with query $\mathsf{S}(1,x)$.
$\leadsto$ Query rule: $\mathsf{Query}(x) \leftarrow \mathsf{S}(1,x)$

Transformed rules:

$$\mathsf{Query}^{f}(x) \leftarrow \mathsf{S}^{bf}(1,x)$$
$$\mathsf{S}^{bf}(x,x) \leftarrow \mathsf{h}(x)$$
$$\mathsf{S}^{bf}(x,y) \leftarrow \mathsf{p}(x,w) \wedge \mathsf{S}^{fb}(v,w) \wedge \mathsf{p}(y,v)$$
$$\mathsf{S}^{fb}(x,x) \leftarrow \mathsf{h}(x)$$
$$\mathsf{S}^{fb}(x,y) \leftarrow \mathsf{p}(x,w) \wedge \mathsf{S}^{fb}(v,w) \wedge \mathsf{p}(y,v)$$

Magic

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?
$\leadsto$ 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
$\leadsto$ 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

$$
\begin{array}{cccccc}
\mathsf{T}^{bf}(x,z) \leftarrow & \mathsf{T}^{bf}(x,y) & \wedge & \mathsf{T}^{bf}(y,z) & & \\
& \Uparrow & \searrow\!\!\!\!\searrow \ \Uparrow & & \searrow\!\!\!\!\searrow & \\
\mathsf{input}^{bf}_{\mathsf{T}} \Rightarrow & \mathsf{sup}_0[x] & & \mathsf{sup}_1[x,y] & & \mathsf{sup}_2[x,z] \Rightarrow \mathsf{output}^{bf}_{\mathsf{T}}
\end{array}
$$

could be expressed using rules:

$$
\begin{aligned}
\mathsf{sup}_0(x) &\leftarrow \mathsf{input}^{bf}_{\mathsf{T}}(x) \\
\mathsf{sup}_1(x,y) &\leftarrow \mathsf{sup}_0(x) \wedge \mathsf{output}^{bf}_{\mathsf{T}}(x,y) \\
\mathsf{sup}_2(x,z) &\leftarrow \mathsf{sup}_1(x,y) \wedge \mathsf{output}^{bf}_{\mathsf{T}}(y,z) \\
\mathsf{output}^{bf}_{\mathsf{T}}(x,z) &\leftarrow \mathsf{sup}_2(x,z)
\end{aligned}
$$

Magic Sets as Simulation of QSQ (2)

Observation: $\text{sup}_0(x)$ and $\text{sup}_2(x, z)$ are redundant. Simpler:

$$\text{sup}_1(x, y) \leftarrow \text{input}_{\mathsf{T}}^{bf}(x) \wedge \text{output}_{\mathsf{T}}^{bf}(x, y)$$
$$\text{output}_{\mathsf{T}}^{bf}(x, z) \leftarrow \text{sup}_1(x, y) \wedge \text{output}_{\mathsf{T}}^{bf}(y, z)$$

Magic Sets as Simulation of QSQ (2)

Observation: $\mathsf{sup}_0(x)$ and $\mathsf{sup}_2(x, z)$ are redundant. Simpler:

$$\mathsf{sup}_1(x, y) \leftarrow \mathsf{input}_{\mathsf{T}}^{bf}(x) \wedge \mathsf{output}_{\mathsf{T}}^{bf}(x, y)$$
$$\mathsf{output}_{\mathsf{T}}^{bf}(x, z) \leftarrow \mathsf{sup}_1(x, y) \wedge \mathsf{output}_{\mathsf{T}}^{bf}(y, z)$$

We still need to "call" subqueries recursively:

$$\mathsf{input}_{\mathsf{T}}^{bf}(y) \leftarrow \mathsf{sup}_1(x, y)$$

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

$$\mathsf{R}^{bf}(x, y) \leftarrow \mathsf{T}^{bbf}(x, a, y)$$

This leads to the following rules using Magic Sets:

$$\mathsf{output}_{\mathsf{R}}^{bf}(x, y) \leftarrow \mathsf{input}_{\mathsf{R}}^{bf}(x) \wedge \mathsf{output}_{\mathsf{T}}^{bbf}(x, a, y)$$
$$\mathsf{input}_{\mathsf{T}}^{bbf}(x, a) \leftarrow \mathsf{input}_{\mathsf{R}}^{bf}(x)$$

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)

$\leadsto$ 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
- QSQ(R)
- 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.

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

$\leadsto$ Different scenarios, different optimal solutions
$\leadsto$ 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)
- Datomic 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
- ...

$\leadsto$ 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