TECHNISCHE UNIVERSITÄT DRESDEN

DATABASE THEORY

Lecture 12: Evaluation of Datalog (2)

Markus Krötzsch

TU Dresden, 30 June 2016

Overview

See course homepage [⇒ link] for more information and materials

Review: Datalog Evaluation

A rule-based recursive query language

$$
\begin{array}{l}
\mathsf{father}(\mathsf{alice},\mathsf{bob}) \\
\mathsf{mother}(\mathsf{alice},\mathsf{carla}) \\
\mathsf{Parent}(x,y) \leftarrow \mathsf{father}(x,y) \\
\mathsf{Parent}(x,y) \leftarrow \mathsf{mother}(x,y) \\
\mathsf{SameGeneration}(x,x) \\
\mathsf{SameGeneration}(x,y) \leftarrow \mathsf{Parent}(x,v) \wedge \mathsf{Parent}(y,w) \wedge \mathsf{SameGeneration}(v,w)
\end{array}
$$

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

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

$$
\begin{array}{rl}
(R1) & \mathsf{T}(x,y) \leftarrow \mathsf{e}(x,y) \\
(R2.1) & \mathsf{T}(x,z) \leftarrow \Delta^i_{\mathsf{T}}(x,y) \wedge \mathsf{T}^i(y,z) \\
(R2.2') & \mathsf{T}(x,z) \leftarrow \mathsf{T}^{i-1}(x,y) \wedge \Delta^i_{\mathsf{T}}(y,z)
\end{array}
$$

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:

$$\mathsf{e}(1,2) \quad \mathsf{e}(2,3) \quad \mathsf{e}(3,4) \quad \mathsf{e}(4,5)$$
$$(R1) \qquad \mathsf{T}(x,y) \leftarrow \mathsf{e}(x,y)$$
$$(R2) \qquad \mathsf{T}(x,z) \leftarrow \mathsf{T}(x,y) \wedge \mathsf{T}(y,z)$$
$$\mathsf{Query}(z) \leftarrow \mathsf{T}(2,z)$$

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

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: 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 note the free/bound parameters in predicates.

Example:

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

- 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:

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

The order of body predicates matters affects the adornment:

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

$\rightsquigarrow$ 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
$\rightsquigarrow$ for adorned relation R^{α}, we use an auxiliary relation $\mathsf{input}^{\alpha}_{\mathsf{R}}$
$\rightsquigarrow$ arity of $\mathsf{input}^{\alpha}_{\mathsf{R}}$ = number of b in α

The result of calling a rule should be the “completed” input, with values for the unbound variables added
$\rightsquigarrow$ for adorned relation R^{α}, we use an auxiliary relation $\mathsf{output}^{\alpha}_{\mathsf{R}}$
$\rightsquigarrow$ arity of $\mathsf{output}^{\alpha}_{\mathsf{R}}$ = 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
$\rightsquigarrow$ bindings required to evaluate rest of rule after the ith body atom
$\rightsquigarrow$ the first set of bindings sup_0 comes from $\mathsf{input}^{\alpha}_{\mathsf{R}}$
$\rightsquigarrow$ the last set of bindings sup_n go to $\mathsf{output}^{\alpha}_{\mathsf{R}}$

Example:

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

- $\mathsf{sup}_0[x]$ 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

Notation we will use:

- 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_i as projection of $\mathsf{sup}^{r}_{i-1} \bowtie A^{\mathcal{I}}_i$
   - 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

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

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

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 $\mathsf{S}(1, x)$.
$\leadsto$ 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 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?

Example:

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

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

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

We still need to "call" subqueries recursively:

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

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.

Example: the following rule is correctly adorned

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

This leads to the following rules using Magic Sets:

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

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

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

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 2015:

- DLV Answer set programming engine with good performance on Datalog programs (commercial)
- LogicBlox Big data analytics platform that uses Datalog rules (commercial)
- Datomic Distributed, versioned database using Datalog as main query language (commercial)

Several RDF (graph data model) DBMS also support Datalog-like rules, usually with limited IDB arity, e.g.:

- OWLIM Disk-backed RDF database with materialisation at load time (commercial)
- RDFox Fast in-memory RDF database with runtime materialisation and updates (academic)

$\leadsto$ 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
- Applications