



TECHNISCHE
UNIVERSITÄT
DRESDEN

DATABASE THEORY

Lecture 15: Datalog Evaluation (2)

Markus Krötzsch

Knowledge-Based Systems

TU Dresden, 5th June 2019

Review: Datalog Evaluation

A rule-based recursive query language

```
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

$$\begin{array}{l} \text{e}(1, 2) \quad \text{e}(2, 3) \quad \text{e}(3, 4) \quad \text{e}(4, 5) \\ (R1) \quad T(x, y) \leftarrow \text{e}(x, y) \\ (R2.1) \quad T(x, z) \leftarrow \Delta_T^i(x, y) \wedge T^i(y, z) \\ (R2.2') \quad T(x, z) \leftarrow T^{i-1}(x, y) \wedge \Delta_T^i(y, z) \end{array}$$

How many body matches do we need to iterate over?

$$\begin{array}{ll} T_P^0 = \emptyset & \text{initialisation} \\ T_P^1 = \{T(1, 2), T(2, 3), T(3, 4), T(4, 5)\} & 4 \times (R1) \\ T_P^2 = T_P^1 \cup \{T(1, 3), T(2, 4), T(3, 5)\} & 3 \times (R2.1) \\ T_P^3 = T_P^2 \cup \{T(1, 4), T(2, 5), T(1, 5)\} & 3 \times (R2.1), 2 \times (R2.2') \\ T_P^4 = T_P^3 = T_P^\infty & 1 \times (R2.1), 1 \times (R2.2') \end{array}$$

In total, we considered 14 matches to derive 11 facts

Semi-Naive Evaluation: Full Definition

In general, a rule of the form

$$H(\vec{x}) \leftarrow e_1(\vec{y}_1) \wedge \dots \wedge e_n(\vec{y}_n) \wedge I_1(\vec{z}_1) \wedge I_2(\vec{z}_2) \wedge \dots \wedge I_m(\vec{z}_m)$$

is transformed into m rules

$$H(\vec{x}) \leftarrow e_1(\vec{y}_1) \wedge \dots \wedge e_n(\vec{y}_n) \wedge \Delta_1^i(\vec{z}_1) \wedge I_2^i(\vec{z}_2) \wedge \dots \wedge I_m^i(\vec{z}_m)$$

$$H(\vec{x}) \leftarrow e_1(\vec{y}_1) \wedge \dots \wedge e_n(\vec{y}_n) \wedge I_1^{i-1}(\vec{z}_1) \wedge \Delta_2^i(\vec{z}_2) \wedge \dots \wedge I_m^i(\vec{z}_m)$$

...

$$H(\vec{x}) \leftarrow e_1(\vec{y}_1) \wedge \dots \wedge e_n(\vec{y}_n) \wedge I_1^{i-1}(\vec{z}_1) \wedge I_2^{i-1}(\vec{z}_2) \wedge \dots \wedge \Delta_m^i(\vec{z}_m)$$

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:

$$\begin{array}{l} e(1, 2) \quad e(2, 3) \quad e(3, 4) \quad e(4, 5) \\ (R1) \quad T(x, y) \leftarrow e(x, y) \\ (R2) \quad T(x, z) \leftarrow T(x, y) \wedge T(y, z) \\ \text{Query}(z) \leftarrow T(2, z) \end{array}$$

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

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 $T(x, z) \leftarrow T(x, y) \wedge T(y, z)$, 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 $T(2, z)$ from the rule $T(x, z) \leftarrow T(x, y) \wedge T(y, z)$, 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:

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

- 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:

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

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

Adornments: Examples

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

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

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

The order of body predicates affects the adornment:

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

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

↪ 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

↪ for adorned relation R^α , we use an auxiliary relation input_R^α

↪ arity of input_R^α = number of b in α

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

↪ for adorned relation R^α , we use an auxiliary relation output_R^α

↪ arity of output_R^α = 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 i th body atom

\leadsto the first set of bindings sup_0 comes from input_R^α

\leadsto the last set of bindings sup_n go to output_R^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 i th body atom

\leadsto the first set of bindings sup_0 comes from input_R^α

\leadsto the last set of bindings sup_n go to output_R^α

Example 15.4:

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

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

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

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

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 input_R^α (that unify with rule head) to sup_0^r
- (2) For each body atom A_1, \dots, A_n , do:
 - If A_i is an EDB atom, compute sup_i^r as projection of $\text{sup}_{i-1}^r \bowtie A_i^I$
 - If A_i is an IDB atom with adorned predicate S^β :
 - (a) Add new bindings from sup_{i-1}^r , combined with constants in A_i , to input_S^β
 - (b) If input_S^β changed, recursively evaluate all rules with head predicate S^β
 - (c) Compute sup_i^r as projection of $\text{sup}_{i-1}^r \bowtie \text{output}_S^\beta$
- (3) Add tuples in sup_n^r to output_R^α

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

QSQR Transformation: Example

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

$$S(x, x) \leftarrow h(x)$$

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

with query $S(1, x)$.

QSQR Transformation: Example

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

$$S(x, x) \leftarrow h(x)$$

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

with query $S(1, x)$.

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

Transformed rules:

QSQR Transformation: Example

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

$$S(x, x) \leftarrow h(x)$$

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

with query $S(1, x)$.

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

Transformed rules:

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

QSQR Transformation: Example

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

$$S(x, x) \leftarrow h(x)$$

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

with query $S(1, x)$.

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

Transformed rules:

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

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

QSQR Transformation: Example

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

$$S(x, x) \leftarrow h(x)$$

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

with query $S(1, x)$.

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

Transformed rules:

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

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

$$S^{bf}(x, y) \leftarrow p(x, w) \wedge S^{fb}(v, w) \wedge p(y, v)$$

QSQR Transformation: Example

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

$$S(x, x) \leftarrow h(x)$$

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

with query $S(1, x)$.

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

Transformed rules:

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

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

$$S^{bf}(x, y) \leftarrow p(x, w) \wedge S^{fb}(v, w) \wedge p(y, v)$$

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

QSQR Transformation: Example

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

$$S(x, x) \leftarrow h(x)$$

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

with query $S(1, x)$.

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

Transformed rules:

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

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

$$S^{bf}(x, y) \leftarrow p(x, w) \wedge S^{fb}(v, w) \wedge p(y, v)$$

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

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

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?

~> 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

$$\begin{array}{ccccccc} T^{bf}(x, z) & \leftarrow & T^{bf}(x, y) & \wedge & T^{bf}(y, z) & & \\ & & \uparrow & & \Downarrow \uparrow & \Downarrow & \\ \text{input}_T^{bf} & \Rightarrow & \text{sup}_0[x] & & \text{sup}_1[x, y] & & \text{sup}_2[x, z] \Rightarrow \text{output}_T^{bf} \end{array}$$

could be expressed using rules:

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

Magic Sets as Simulation of QSQ (2)

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

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

Magic Sets as Simulation of QSQ (2)

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

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

We still need to “call” subqueries recursively:

$$\text{input}_T^{bf}(y) \leftarrow \text{sup}_1(x, y)$$

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:

$$\text{output}_R^{bf}(x, y) \leftarrow \text{input}_R^{bf}(x) \wedge \text{output}_T^{bbf}(x, a, y)$$

$$\text{input}_T^{bbf}(x, a) \leftarrow \text{input}_R^{bf}(x)$$

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)

→ 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

↪ Different scenarios, different optimal solutions

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

Datalog Implementation in Practice

Dedicated Datalog engines as of 2018 (incomplete):

- **RDFFox** 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
- **SocialLite** 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
- ...

~> 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