TECHNISCHE UNIVERSITÄT DRESDEN

DATABASE THEORY

Lecture 15: Datalog Evaluation (2) / Graph Databases

Markus Krötzsch

Knowledge-Based Systems

TU Dresden, 13th June 2023

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

Common bottom-up technique: semi-naive evaluation

Strategy for top-down methods: QSQR

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
$\leadsto$ can we just implement this in Datalog?

Example 15.1: The QSQ information flow

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

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:

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

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 15.2: 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:

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

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

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 is often better, especially 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)
- The reduction in inferences is not huge enough to justify the significant extra effort for magic sets (more joins, more fragmented data, more rules, more iterations)

$\leadsto$ semi-naive evaluation is more 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? How to update caches (view maintenance)?
- Is intra-query parallelism desirable? On which level and for how many 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? How to update caches (view maintenance)?
- Is intra-query parallelism desirable? On which level and for how many CPUs?
- ...

Further decisions relate to the supported Datalog dialect:

- Should negation be supported? In which cases?
- Will there be datatypes? Which? Type system?
- Aggregate functions and built-ins?
- Other logical language exstensions (disjunction, existential quantifiers, function symbols, ...)?
- ...

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 2023 (incomplete):

- Nemo Fast in-memory Datalog materialisation, various language extensions and bindings (free, Rust, developed at TU Dresden)
- VLog/Rulewerk Fast in-memory Datalog materialisation with bindings to several databases, including RDF and RDBMS (free, C++/Java, co-developed at TU Dresden)
- Soufflé Fast in-memory Datalog engine for program analysis (free, C++)
- Graal In-memory rule engine with RDBMS bindings (free, Java)
- Gringo Fast Datalog-based grounder for answer set programming (free, C++)
- RDFox Fast in-memory RDF database with runtime materialisation and updates (commercial)
- Vadalog Closed-source engine with several extensions (commercial)
- Llunatic PostgreSQL-based implementation of a rule engine (free, discontinued)
- SociaLite and EmptyHeaded Datalog-based languages and engines for social network analysis
- DeepDive Data analysis platform with support for Datalog-based language “DDlog”
- Datomic Distributed, versioned database using Datalog as main query language (commercial)
- LogicBlox Big data analytics platform that uses Datalog rules (commercial, discontinued)
- 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

Graph Databases

Graph Databases

Our original motivation for going from FO queries to Datalog:
Reachability of nodes in a (directed) graph $\leadsto$ let's focus on graphs

Graph database: a DBMS that supports "graphs" as its datamodel

There are many kinds of graphs:

- Directed or undirected?
- Labelled or unlabelled edges/nodes?
- What kinds of labels? Datatypes?
- Parallel edges (multi-graphs)? With same label?
- One graph or several graphs per database?

Two types of graph database models dominate the market today: Resource Description Framework (RDF) and Property Graph

Resource Description Framework (RDF)

RDF is a W3C standard for representing linked data on the Web

- Directed labelled graph; nodes are identified by their labels
- Labels are URIs or datatype literals
- Multiple parallel edges only when using different edge labels
- Supports multiple graphs in one database
- W3C standard; implementations for many programming languages
- Datatype support based on W3C XML Schema datatypes
- Graphs can be exchanged in many standard syntax formats

Property Graph

Property Graph is a popular data model of many graph databases

- Directed labelled multi-graph; labels do not identify nodes
- "Labels" can be lists of attribute-value pairs
- Multiple parallel edges with the exact same labels are possible
- No native multi-graph support (could be simulated with additional attributes)
- No standard definition of technical details; most common implementation: Tinkerpop/Blueprints API (Java)
- Datatype support varies by implementation
- No standard syntax for exchanging data

Representing Graphs

Graphs (of any type) are usually viewed as sets of edges

- RDF: triples of form subject-predicate-object
  - When managing multiple graphs, each triple is extended with a fourth component (graph ID) $\leadsto$ quads
  - RDF databases are sometimes still called "triple stores", although most modern systems effectively store quads
- Property Graph: edge objects with attribute lists
  - represented by Java objects in Blueprints

Graphs can be stored in relational databases

- RDF: table Triple[Subject,Predicate,Object]
- Property Graph: tables Edge[SourceId,EdgeId,TargetId] and Attributes[Id,Attribute,Value]

Representing Data in Graphs

Property Graphs can represent RDF:

- use additional nodes to represent triples, and connect them to subject/predicate/object nodes[1]
- use attributes to store RDF ids (URIs) and data values (literals)
- use key constraints to ensure that no two distinct nodes can have same label

[1] Using PG edge labels is not enough, since RDF models frequently need to use edge labels as subjects or objects of other triples, and PG does not support this

Representing Data in Graphs

Property Graphs can represent RDF:

- use additional nodes to represent triples, and connect them to subject/predicate/object nodes[1]
- use attributes to store RDF ids (URIs) and data values (literals)
- use key constraints to ensure that no two distinct nodes can have same label

RDF can represent Property Graphs:

- use additional nodes to represent Property Graph edges, and connect them with their source and target nodes
- use RDF triples with special predicates to represent attributes

Either model can also represent hypergraphs/RDBs (exercise)

$\leadsto$ all models can represent all data in principle
$\leadsto$ supported query features and performance will vary

[1] Using PG edge labels is not enough, since RDF models frequently need to use edge labels as subjects or objects of other triples, and PG does not support this

Querying Graphs

Preferred query language depends on graph model

- RDF: W3C SPARQL query language
- Property Graph: no uniform approach to data access
  - many tools prefer API access over a query language
  - proprietary query languages, e.g., “Cypher” for Neo4j

However, there are some common basics in almost all cases:[1]

- Conjunctive queries
- Regular path queries

[1]Might not be true for Cypher, which – in contrast to most other database query languages – is based on a variant of graph isomorphism rather than homomorphism; and which supports only specific path expressions

Conjunctive Queries over Graphs

Basic descriptions of local patterns in a graph

Formally, it suffices to say:

"CQs over RDF correspond to CQs over relational databases with a single table Triple[Subject,Predicate,Object]"

(and analogously for Property Graphs)

- All complexity results for query answering and optimisation carry over from RDBs (in particular, restricting to graphs does not make anything simpler)
- Details of representation of data in tables do not matter
- CQs are restricted to local patterns (no reachability ...)

Regular Path Queries

Idea: use regular expressions to navigate over paths

Let's consider a simplified graph model, where a graph is given by:

- Set of nodes N (without additional labels)
- Set of edges E, labelled by a function $\lambda : E \to L$, where L is a finite set of labels

Definition 15.3: A regular expression over a set of labels L is an expression of the following form:

$$E ::= L \mid (E \circ E) \mid (E + E) \mid E^*$$

A regular path query (RPQ) is an expression of the form $E(s, t)$, where E is a regular expression and s and t are terms (constants or variables).

Semantics of Regular Path Queries

As usual, a regular expression E matches a word $w = \ell_1 \cdots \ell_n$ if any of the following conditions is satisfied:

- $E \in L$ is a label and $w = E$.
- $E = (E_1 \circ E_2)$ and there is $i \in \{0, \ldots, n\}$ such that E_1 matches $\ell_1 \cdots \ell_i$ and E_2 matches $\ell_{i+1} \cdots \ell_n$ (the words matched by E_1 and E_2 can be empty if $i = 0$ or $i = n$, respectively).
- $E = (E_1 + E_2)$ and w is matched by E_1 or by E_2
- $E = E_1^*$ and w has the form $w_1 w_2 \cdots w_m$ for $n \geq 0$, where each word w_i is matched by E_1

Definition 15.4: Let a and b be constants and x and y be variables. An RPQ $E(a, b)$ is entailed by a graph G if there is a directed path from node a to node b that is labelled by a word matched by E. The answers to RPQs $E(x, y)$, $E(x, b)$, and $E(a, y)$ are defined in the obvious way.

Extending the Expressive Power of RPQs

Regular path queries can be used to express typical reachability queries, but are still quite limited $\leadsto$ extensions

2-Way Regular Path Queries (2RPQs)

- For every label $\ell \in L$, also introduce a converse label ℓ^-
- Allow converse labels in regular expressions
- Matched paths can follow edges forwards or backwards

Conjunctive Regular Path Queries (CRPQs)

- Extend conjunctive queries with RPQs
- RPQs can be used like binary query atoms
- Obvious semantics

Conjunctive 2-Way Regular Path Queries (C2RPQs) combine both extensions

C2RPQs: Examples

All ancestors of Alice:

$$((\text{father} + \text{mother}) \circ (\text{father} + \text{mother})^*)(\text{alice}, y)$$

C2RPQs: Examples

All ancestors of Alice:

$$((\mathsf{father} + \mathsf{mother}) \circ (\mathsf{father} + \mathsf{mother})^*)(\mathsf{alice}, y)$$

People with finite Erdös number:

$$(\mathsf{authorOf} \circ \mathsf{authorOf}^-)^*(x, \mathsf{paulErdös})$$

C2RPQs: Examples

All ancestors of Alice:

$$((\mathsf{father} + \mathsf{mother}) \circ (\mathsf{father} + \mathsf{mother})^*)(\mathsf{alice}, y)$$

People with finite Erdös number:

$$(\mathsf{authorOf} \circ \mathsf{authorOf}^-)^*(x, \mathsf{paulErdös})$$

Pairs of stops connected by tram lines 3 and 8:

$$(\mathsf{nextStop3} \circ \mathsf{nextStop3}^*)(x, y) \wedge (\mathsf{nextStop8} \circ \mathsf{nextStop8}^*)(x, y)$$

Summary and Outlook

Several implementation techniques for Datalog

- naive evaluation (bottom-up, not goal-directed)
- semi-naive evaluation (bottom-up, not goal-directed)
- Query-Subquery (QSQ) approach (top-down, goal-directed)
- Magic Sets (bottom-up, goal-directed)

Graph databases as an important class of “noSQL” databases

Two main data models

- Resource Description Framework (RDF)
- Property Graph

Conceptual graph query language: regular path queries

Next topics:

- More about path queries (complexity . . .)
- Dependencies