Review: Datalog Evaluation

A rule-based recursive query language

\[
\begin{align*}
\text{father}(\text{alice}, \text{bob}) \\
\text{mother}(\text{alice}, \text{carla}) \\
\text{Parent}(x, y) &\leftarrow \text{father}(x, y) \\
\text{Parent}(x, y) &\leftarrow \text{mother}(x, y) \\
\text{SameGeneration}(x, y) &\leftarrow \text{Parent}(x, y) \land \text{Parent}(y, w) \land \text{SameGeneration}(y, w)
\end{align*}
\]

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

\[
\begin{align*}
&e(1, 2) \quad e(2, 3) \quad e(3, 4) \quad e(4, 5) \\
&(R1) \quad T(x, y) \leftarrow e(x, y) \\
&(R2.1) \quad T(x, z) \leftarrow \text{I}_1 \\
&(R2.2') \quad T(x, z) \leftarrow T^{x-1}(x, y) \land \text{I}_2(y, z)
\end{align*}
\]

In total, we considered 14 matches to derive 11 facts

How many body matches do we need to iterate over?

\[
\begin{align*}
T_P^0 &= 0 & \text{initialisation} \\
T_P^1 &= \{T(1, 2), T(2, 3), T(3, 4), T(4, 5)\} & 4 \times (R1) \\
T_P^2 &= T_P^1 \cup \{T(1, 3), T(2, 4), T(3, 5)\} & 3 \times (R2.1) \\
T_P^3 &= T_P^2 \cup \{T(1, 4), T(2, 5), T(1, 5)\} & 3 \times (R2.1), 2 \times (R2.2') \\
T_P^n &= T_{P, n} & 1 \times (R2.1), 1 \times (R2.2')
\end{align*}
\]

In total, we considered 14 matches to derive 11 facts

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:**

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

Query(z) \leftarrow T(2, z)

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 \(T(x, z) \leftarrow T(x, y) \land T(y, z)\), 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:**

\[
T^B_f(x, z) \leftarrow T^B_f(x, y) \land T^B_f(y, z)
\]

- 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:

- \( R^{hh}(x, y, z) \leftarrow R^{hh}(x, y, v) \land R^{hh}(x, v, z) \)
- \( R^{hy}(x, y, z) \leftarrow R^{hy}(x, y, v) \land R^{hy}(x, v, z) \)

The order of body predicates affects the adornment:

- \( S^{hy}(x, y, z) \leftarrow T^{hy}(x, v) \land T^{hy}(y, w) \land R^{hy}(v, w, z) \)
- \( S^{hy}(x, y, z) \leftarrow R^{hy}(v, w, z) \land T^{hy}(x, v) \land T^{hy}(y, w) \)

\( \leadsto \) 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 \( R^n \), we use an auxiliary relation \( \text{input}^\alpha_R \)
- \( \leadsto \) arity of \( \text{input}^\alpha_R = \text{number of } b \text{ in } \alpha \)

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

- \( \leadsto \) for adorned relation \( R^n \), we use an auxiliary relation \( \text{output}^\alpha_R \)
- \( \leadsto \) arity of \( \text{output}^\alpha_R = \text{arity of } R (= \text{length of } \alpha) \)

Example 15.4:

- \( T^{hy}(x, z) \leftarrow T^{hy}(x, y) \land T^{hy}(y, z) \)
- \( \text{input}^\beta_T \Rightarrow \text{sup}^\beta_1[x] \uparrow \uparrow \downarrow \downarrow \text{sup}^\beta_1[x, y] \downarrow \text{sup}^\beta_2[x, z] \Rightarrow \text{output}^\beta_T \)

- \( \text{sup}^\beta_1[x] \) is copied from \( \text{input}^\beta_T [x] \) (with some exceptions, see exercise)
- \( \text{sup}^\beta_1[x, y] \) is obtained by joining tables \( \text{sup}^\beta_1[x] \) and \( \text{output}^\beta_T [x, y] \)
- \( \text{sup}^\beta_2[x, z] \) is obtained by joining tables \( \text{sup}^\beta_1[x, y] \) and \( \text{output}^\beta_T [y, z] \)
- \( \text{output}^\beta_T [x, z] \) is copied from \( \text{sup}^\beta_2[x, z] \)

(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:

- 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 \( R^\alpha \); current values of all QSQ relations

1. Copy tuples \( \text{input}^\alpha_r \) (that unify with rule head) to \( \text{sup}^\alpha_r \)
2. For each body atom \( A_1, \ldots, A_n \), do:
   - If \( A_i \) is an EDB atom, compute \( \text{sup}^\alpha_i \) as projection of \( \text{sup}^\alpha_{i-1} \) \( \bowtie \) \( \text{input}^\beta_S \)
   - If \( A_i \) is an IDB atom with adorned predicate \( S^\beta \):
     (a) Add new bindings from \( \text{sup}^\alpha_{i-1} \), 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 \( \text{sup}^\alpha_i \) as projection of \( \text{sup}^\alpha_{i-1} \) \( \bowtie \) \( \text{output}^\beta_S \)
3. Add tuples in \( \text{sup}^\alpha_n \) to \( \text{output}^\alpha_R \)

Markus Krötzsch, 5th June 2019
Database Theory slide 13 of 27

**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 \( P^\alpha \):
   - Turn the query \( q[\vec{x}] \) into an adorned rule \( \text{Query} \)
   - Recursively create adorned rules from rules in \( P \) for all adorned predicates in \( P^\alpha \).
2. Initialise all auxiliary relations to empty sets.
3. Evaluate the rule \( \text{Query} \) repeatedly until no new tuples are added to any QSQ relation.
4. Return \( \text{output}^\alpha_{\text{Query}} \).

Markus Krötzsch, 5th June 2019
Database Theory slide 14 of 27

**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 \( S(1, x) \).

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

Transformed rules:

\[
\text{Query}^\beta(x) \leftarrow S^\beta(1, x)
\]
\[
S^\beta(x, x) \leftarrow h(x)
\]
\[
S^\beta(x, y) \leftarrow p(x, w) \land S^\beta(v, w) \land p(y, v)
\]
\[
S^\beta(x, x) \leftarrow h(x)
\]
\[
S^\beta(x, y) \leftarrow p(x, w) \land S^\beta(v, w) \land p(y, v)
\]

Markus Krötzsch, 5th June 2019
Database Theory slide 15 of 27

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?

→ 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?

Example 15.5: The QSQ information flow

\[
\begin{align*}
T^{bf}(x, z) & \leftarrow T^{bf}(x, y) \land T^{bf}(y, z) \\
\text{input}^{bf} & \Rightarrow \text{sup}_0[x] \quad \text{sup}_1[x, y] \quad \text{sup}_2[x, z] \Rightarrow \text{output}^{bf}
\end{align*}
\]

could be expressed using rules:

\[
\begin{align*}
\text{sup}_0(x) & \leftarrow \text{input}^{bf}(x) \\
\text{sup}_1(x, y) & \leftarrow \text{sup}_0(x) \land \text{output}^{bf}(x, y) \\
\text{sup}_2(x, z) & \leftarrow \text{sup}_1(x, y) \land \text{output}^{bf}(y, z) \\
\text{output}^{bf}(x, z) & \leftarrow \text{sup}_2(x, z)
\end{align*}
\]

A Note on Constants

Constants in rule bodies must lead to bindings in the subquery

Example 15.6: The following rule is correctly adorned

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

This leads to the following rules using Magic Sets:

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

Note that we do not need to use auxiliary predicates \text{sup}_0 or \text{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)

\[ \Rightarrow \text{semi-naive evaluation is still very common in practice} \]

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)

...
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)

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

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

~ Extremely diverse tools for very different requirements