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, y)
  - 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

e(1, 2) e(2, 3) e(3, 4) e(4, 5)
(R1) T(x, y) ← e(x, y)
(R2.1) T(x, z) ← Δ₁(x, y) ∧ T₁(y, z)
(R2.2’) T(x, z) ← T⁻¹(x, y) ∧ Δ₂(y, z)

How many body matches do we need to iterate over?

\[
\begin{align*}
T₀ &= \emptyset & \text{(initialisation)} \\
T₁ &= \{T(1, 2), T(2, 3), T(3, 4), T(4, 5)\} & 4 \times (R1) \\
T₂ &= T₁ \cup \{T(1, 3), T(2, 4), T(3, 5)\} & 3 \times (R2.1) \\
T₃ &= T₂ \cup \{T(1, 4), T(2, 5), T(1, 5)\} & 3 \times (R2.1), 2 \times (R2.2’)
\end{align*}
\]

\[
\begin{align*}
T₄ &= T₃ = T^∞ & 1 \times (R2.1), 1 \times (R2.2’)
\end{align*}
\]

In total, we considered 14 matches to derive 11 facts
Semi-Naive Evaluation: Full Definition

In general, a rule of the form

\[ H(\vec{x}) \leftarrow e_1(\vec{y}_1) \land \ldots \land e_n(\vec{y}_n) \land I_1(\vec{z}_1) \land I_2(\vec{z}_2) \land \ldots \land I_m(\vec{z}_m) \]

is transformed into \( m \) rules:

\[ H(\vec{x}) \leftarrow e_1(\vec{y}_1) \land \ldots \land e_n(\vec{y}_n) \land I_1(\vec{z}_1) \land I_2(\vec{z}_2) \land \ldots \land I_m(\vec{z}_m) \]
\[ H(\vec{x}) \leftarrow e_1(\vec{y}_1) \land \ldots \land e_n(\vec{y}_n) \land \Delta_{I_1}^{-1}(\vec{z}_1) \land I_2(\vec{z}_2) \land \ldots \land I_m(\vec{z}_m) \]
\[ \ldots \]
\[ H(\vec{x}) \leftarrow e_1(\vec{y}_1) \land \ldots \land e_n(\vec{y}_n) \land \Delta_{I_m}^{-1}(\vec{z}_m) \land I_1(\vec{z}_1) \land I_2(\vec{z}_2) \land \ldots \land \Delta_{I_m}^{-1}(\vec{z}_m) \]

Advantages and disadvantages:

- Huge improvement over naive evaluation
- Some redundant computations remain (see example)
- Some overhead for implementation (store level of entailments)

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 \( T(2, z) \) 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 note the free/bound parameters in predicates.

Example:
\[
T^{bf}(x, z) \leftarrow T^{bf}(x, y) \land T^{bf}(y, z)
\]

- 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)

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 \( R^a \), we use an auxiliary relation \( \text{input}^{a}_{R} \)
~ arity of \( \text{input}^{a}_{R} \) = number of \( b \) in \( a \)

The result of calling a rule should be the “completed” input, with values for the unbound variables added
~ for adorned relation \( R^a \), we use an auxiliary relation \( \text{output}^{a}_{R} \)
~ arity of \( \text{output}^{a}_{R} \) = arity of \( R \) (= length of \( a \))

Adornments: Examples

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

\[
R^{bbb}(x, y, z) \leftarrow R^{bf}(x, y, v) \land R^{bb}(x, v, z)
\]

\[
R^{bf}(x, y, z) \leftarrow R^{bf}(x, y, v) \land R^{bb}(x, v, z)
\]

The order of body predicates matters affects the adornment:

\[
S^{bf}(x, y, z) \leftarrow T^{bf}(x, v) \land T^{bf}(y, w) \land R^{bb}(v, w, z)
\]

\[
S^{bf}(x, y, z) \leftarrow R^{bf}(v, w, z) \land T^{bf}(x, v) \land T^{bf}(y, w)
\]

~ For optimisation, some orders might be better than others

Auxiliary Relations for QSQ (2)

When evaluating body atoms from left to right, we use supplementary relations \( \text{sup}_i \)
~ bindings required to evaluate rest of rule after the \( i \)th body atom
~ the first set of bindings \( \text{sup}_0 \) comes from \( \text{input}^{a}_{R} \)
~ the last set of bindings \( \text{sup}_n \) go to \( \text{output}^{a}_{R} \)

Example:

\[
T^{bf}(x, z) \leftarrow T^{bf}(x, y) \land T^{bf}(y, z)
\]

\[
\text{input}^{bf}_T \Rightarrow \text{sup}_0[x] \uparrow \text{sup}_1[x, y] \downarrow \text{sup}_2[x, z] \Rightarrow \text{output}^{bf}_T
\]

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

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 $Query^\alpha \leftarrow q[\vec{x}]$
   - 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 $Query^\alpha \leftarrow q[\vec{x}]$.
   Repeat until no new tuples are added to any QSQR relation.
4. Return output $\text{output}^{\alpha}_{\text{Query}}$

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 input $^\alpha_r$ (that unify with rule head) to sup $^\alpha_r$
2. For each body atom $A$ do:
   - If $A_i$ is an EDB atom, compute sup $^\alpha_i$ as projection of sup $^\alpha_i$ $\leadsto A_i^\alpha$
   - If $A_i$ is an IDB atom with adorned predicate $S^\beta$:
     a) Add new bindings from sup $^\alpha_{i-1}$, combined with constants in $A_i$, to input $^\beta_S$
     b) If input $^\beta_S$ changed, recursively evaluate all rules with head predicate $S^\beta$
     c) Compute sup $^\alpha_i$ as projection of sup $^\alpha_{i-1} \leadsto \text{output}^\beta_S$
3. Add tuples in sup $^\alpha_n$ to output $^\alpha_R$
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?
\[ \leadsto \text{yes, by magic} \]

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 \[ \leadsto \text{can we just implement this in Datalog?} \]

Example:

\[
\begin{align*}
T^{bf}(x, z) & \leftarrow T^{bf}(x, y) \land T^{bf}(y, z) \\
\text{input}^{bf} & \Rightarrow \text{sup}_{0}[x] \quad \text{sup}_{1}[x, y] \quad \text{sup}_{2}[x, z] \Rightarrow \text{output}^{bf}
\end{align*}
\]

Could be expressed using rules:

\[
\begin{align*}
\text{sup}_{0}(x) & \leftarrow \text{input}^{bf}(x) \\
\text{sup}_{1}(x, y) & \leftarrow \text{sup}_{0}(x) \land \text{output}^{bf}_{T}(x, y) \\
\text{sup}_{2}(x, z) & \leftarrow \text{sup}_{1}(x, y) \land \text{output}^{bf}_{T}(y, z) \\
\text{output}^{bf}_{T}(x, z) & \leftarrow \text{sup}_{2}(x, z)
\end{align*}
\]

A Note on Constants

Constants in rule bodies must lead to bindings in the subquery.

Example: the following rule is correctly adorned

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

This leads to the following rules using Magic Sets:

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

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)

\(\Rightarrow\) 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

\(\Rightarrow\) Different scenarios, different optimal solutions
\(\Rightarrow\) 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)

\(\Rightarrow\) 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