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) ← Parent(x, y) ∧ Parent(y, w) ∧ SameGeneration(y, 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: 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

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

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

\[ \ldots \]

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:**

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

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:**

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

- 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:

\[ R^{\text{hh}}(x, y, z) \leftarrow R^{\text{hh}}(x, y, v) \land R^{\text{hh}}(x, v, z) \]
\[ R^{\text{ff}}(x, y, z) \leftarrow R^{\text{ff}}(x, y, v) \land R^{\text{ff}}(x, v, z) \]

The order of body predicates affects the adornment:

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

\( \leadsto \) 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 \( \text{sup}_i \)
\( \leadsto \) bindings required to evaluate rest of rule after the \( i \)th body atom
\( \leadsto \) the first set of bindings \( \text{sup}_1 \) comes from \( \text{input}^1_R \)
\( \leadsto \) the last set of bindings \( \text{sup}_n \) go to \( \text{output}^n_R \)

**Example 15.4:**

\[ T^{\text{ff}}(x, z) \leftarrow T^{\text{ff}}(x, y) \land T^{\text{ff}}(y, z) \]
\[ \text{input}^f_T \Rightarrow \text{sup}_0[x] \quad \text{sup}_1[x, y] \quad \text{sup}_2[x, z] \Rightarrow \text{output}^f_T \]

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

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 \( \text{input}^n_R \)
\( \leadsto \) arity of \( \text{input}^n_R \) = number of \( b \) 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 \( \text{output}^n_R \)
\( \leadsto \) arity of \( \text{output}^n_R \) = arity of \( R (= \text{length of } \alpha) \)

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 \( A^f \) 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 input\(^0\) \( R \) (that unify with rule head) to sup\(^0\)
(2) For each body atom \( A_1, \ldots, A_n \), do:
   - If \( A_i \) is an EDB atom, compute sup\(^i\) as projection of sup\(^i-1\) \( \Join \leftarrow \leftarrow \) \( A_i \)
   - If \( A_i \) is an IDB atom with adorned predicate \( S \) \( \beta \):
     (a) Add new bindings from sup\(^i-1\), combined with constants in \( A_i \), to input\(^i\) \( S \)
     (b) If input\(^i\) \( S \) changed, recursively evaluate all rules with head predicate \( S \) \( \beta \)
     (c) Compute sup\(^i\) as projection of sup\(^i-1\) \( \Join \leftarrow \leftarrow \) output\(^i\)
(3) Add tuples in sup\(^n\) to output\(^R\)

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 Query\(\vec{f} \leftarrow \vec{x} \)
   - 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 Query\(\vec{f} \leftarrow \vec{x} \).
Repeat until no new tuples are added to any QSQ relation.
(4) Return output\(\vec{f} \leftarrow \vec{x} \) Query.

QSQR Transformation: Example

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

\[
\begin{align*}
  S(x, x) & \leftarrow h(x) \\
  S(x, y) & \leftarrow p(x, w) \land S(v, w) \land p(y, v)
\end{align*}
\]

with query \( S(1, x) \).

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

Transformed rules:

\[
\begin{align*}
  \text{Query}'(x) & \leftarrow S'^{(1)}(1, x) \\
  S'^{(x, x)} & \leftarrow h(x) \\
  S'^{(x, y)} & \leftarrow p(x, w) \land S'^{(v, w)} \land p(y, v) \\
  S'^{(x, x)} & \leftarrow h(x) \\
  S'^{(x, y)} & \leftarrow p(x, w) \land S'^{(v, w)} \land p(y, v)
\end{align*}
\]

Markus Krötzsch, 26th June 2018
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 15.5: The QSQ information flow
\[
\begin{align*}
T^\text{bf}(x,z) & \leftarrow T^\text{bf}(x,y) \land T^\text{bf}(y,z) \\
\text{input}^\text{bf} & \Rightarrow \text{sup}_0[x] \quad \text{sup}_1[x,y] \quad \text{sup}_2[x,z] \Rightarrow \text{output}^\text{bf}
\end{align*}
\]

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

We still need to “call” subqueries recursively:
\[
\begin{align*}
\text{input}^\text{bf}_1(y) & \leftarrow \text{sup}_1(x,y) \\
\text{input}^\text{bf}_2(y) & \leftarrow \text{sup}_2(x,y)
\end{align*}
\]

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 15.6: The following rule is correctly adorned
\[
R^\text{bf}(x,y) \leftarrow T^\text{bbf}(x,a,y)
\]

This leads to the following rules using Magic Sets:
\[
\begin{align*}
\text{output}^\text{bf}_3(x,y) & \leftarrow \text{input}^\text{bf}_3(x,y) \land \text{output}^\text{bf}_2(x,a,y) \\
\text{input}^\text{bf}_4(x,a) & \leftarrow \text{input}^\text{bf}_2(x)
\end{align*}
\]

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)

→ 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)

. . .

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.

- **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

Arrowed different scenarios, different optimal solutions
~> Not all implementations are complete (e.g., Prolog)

### 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 EmptyHeade** Data-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?)
- **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

Arrowed 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