Overview

1. Introduction | Relational data model
2. First-order queries
3. Complexity of query answering
4. Complexity of FO query answering
5. Conjunctive queries
6. Tree-like conjunctive queries
7. Query optimisation
8. Conjunctive Query Optimisation / First-Order Expressiveness
9. First-Order Expressiveness / Introduction to Datalog
10. Expressive Power and Complexity of Datalog
11. Optimisation and Evaluation of Datalog
12. Evaluation of Datalog (2)
13. Graph Databases and Path Queries
14. Outlook: database theory in practice

See course homepage [⇒ link] for more information and materials
A rule-based recursive query language

father(alice, bob)
mother(alice, carla)

\[
\text{Parent}(x, y) \leftarrow \text{father}(x, y)
\]

\[
\text{Parent}(x, y) \leftarrow \text{mother}(x, y)
\]

SameGeneration(x, x)

SameGeneration(x, y) \leftarrow \text{Parent}(x, v) \land \text{Parent}(y, w) \land \text{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{align*}
&\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^i_T(x, y) \land T^i(y, z) \\
&(R2.2') \quad T(x, z) \leftarrow T^{i-1}(x, y) \land \Delta^i_T(y, z)
\end{align*}

How many body matches do we need to iterate over?

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

In total, we considered 14 matches to derive 11 facts
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

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

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

\[ \ldots \]

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

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:

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

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

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: 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.
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)
Adornments: Examples

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

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

\[
R^{fbf}(x, y, z) \leftarrow R^{fbf}(x, y, v) \land R^{bbf}(x, v, z)
\]
Adornments: Examples

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

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

The order of body predicates matters affects the adornment:

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

\(\sim\) 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

\[ \sim \] for adorned relation \( R^\alpha \), we use an auxiliary relation \( \text{input}_R^\alpha \)

\[ \sim \] arity of \( \text{input}_R^\alpha = \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

\[ \sim \] for adorned relation \( R^\alpha \), we use an auxiliary relation \( \text{output}_R^\alpha \)

\[ \sim \] arity of \( \text{output}_R^\alpha = \text{arity of } R \text{ (} = \text{length of } \alpha \text{)} \]
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 $input^\alpha_R$

$\leadsto$ the last set of bindings $sup_n$ go to $output^\alpha_R$
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 $\text{sup}_0$ comes from input$^\alpha_R$

$\leadsto$ the last set of bindings $\text{sup}_n$ go to output$^\alpha_R$

Example:

$$
\text{T}^{bf}(x, z) \leftarrow \text{T}^{bf}(x, y) \land \text{T}^{bf}(y, z)
\uparrow \quad \uparrow \quad \uparrow \quad \downarrow
\text{input}^{bf}_T \Rightarrow \text{sup}_0[x] \quad \text{sup}_1[x, y] \quad \text{sup}_2[x, z] \Rightarrow \text{output}^{bf}_T
$$

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

Markus Krötzsch, 30 June 2016

Database Theory

slide 15 of 43
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)

\[ \text{there are many strategies for implementing this general scheme} \]
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)

\( \implies \) there are many strategies for implementing this general scheme

Notation we will use:

- for an EDB atom \( A \), we write \( A^T \) 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$^\alpha_R$ (that unify with rule head) to sup$^r_0$

(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$^{r_{i-1}}_i \bowtie A^I_i$
   - If $A_i$ is an IDB atom with adorned predicate $S^\beta$:
     (a) Add new bindings from sup$^{r_{i-1}}_i$, 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$^i$ as projection of sup$^{r_{i-1}}_i \bowtie$ output$^\beta_S$

(3) Add tuples in sup$^r_n$ to output$^\alpha_R$
QSQR Algorithm

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

(4) Return output $^{ff\ldots f}_{\text{Query}}$
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 \( S(1, x) \).
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 \( S(1, x) \).

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

Transformed rules:
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: \( \text{Query}(x) \leftarrow S(1, x) \)

Transformed rules:

\[
\text{Query}^f(x) \leftarrow S^{bf}(1, x)
\]
QSQR Transformation: Example

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

\[
S(x,x) \leftarrow h(x)
\]

\[
S(x,y) \leftarrow p(x,w) \land S(v,w) \land p(y,v)
\]

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

\(~\) 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)

\[
S(x, x) \leftarrow h(x)
\]
\[
S(x, y) \leftarrow p(x, w) \land S(v, w) \land p(y, v)
\]

with query $S(1, x)$.

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

Transformed rules:

\[
\text{Query}^f(x) \leftarrow S^{bf}(1, x)
\]
\[
S^{bf}(x, x) \leftarrow h(x)
\]
\[
S^{bf}(x, y) \leftarrow p(x, w) \land S^{fb}(v, w) \land p(y, v)
\]
QSQR Transformation: Example

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

\[
S(x, x) \leftarrow h(x)
\]
\[
S(x, y) \leftarrow p(x, w) \land S(v, w) \land p(y, v)
\]

with query $S(1, x)$.

$\leadsto$ Query rule: Query$(x) \leftarrow S(1, x)$

Transformed rules:

\[
\text{Query}^f(x) \leftarrow S^{bf}(1, x)
\]
\[
S^{bf}(x, x) \leftarrow h(x)
\]
\[
S^{bf}(x, y) \leftarrow p(x, w) \land S^{fb}(v, w) \land p(y, v)
\]
\[
S^{fb}(x, x) \leftarrow h(x)
\]
QSQR Transformation: Example

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

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

with query $S(1, x)$.

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

Transformed rules:

$$\text{Query}^f(x) \leftarrow S^{bf}(1, x)$$
$$S^{bf}(x, x) \leftarrow h(x)$$
$$S^{bf}(x, y) \leftarrow p(x, w) \land S^{fb}(v, w) \land p(y, v)$$
$$S^{fb}(x, x) \leftarrow h(x)$$
$$S^{fb}(x, y) \leftarrow p(x, w) \land S^{fb}(v, w) \land p(y, v)$$
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?

\[ \rightarrow \text{yes, by magic} \]

**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

→ 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.

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

Example:

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

\[\uparrow \quad \downarrow \quad \uparrow \quad \downarrow\]

\[\text{input}_{\text{T}}^{bf} \Rightarrow \text{sup}_{0}[x] \quad \text{sup}_{1}[x, y] \quad \text{sup}_{2}[x, z] \Rightarrow \text{output}_{\text{T}}^{bf}\]

Could be expressed using rules:

\[\text{sup}_{0}(x) \leftarrow \text{input}_{\text{T}}^{bf}(x)\]

\[\text{sup}_{1}(x, y) \leftarrow \text{sup}_{0}(x) \land \text{output}_{\text{T}}^{bf}(x, y)\]

\[\text{sup}_{2}(x, z) \leftarrow \text{sup}_{1}(x, y) \land \text{output}_{\text{T}}^{bf}(y, z)\]

\[\text{output}_{\text{T}}^{bf}(x, z) \leftarrow \text{sup}_{2}(x, z)\]
Magic Sets as Simulation of QSQ (2)

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

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

We still need to “call” subqueries recursively:

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

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: the following rule is correctly adorned

$$R^{bf}(x, y) \leftarrow T^{bbf}(x, a, z)$$

This leads to the following rules using Magic Sets:

$$\text{output}_{R}^{bf}(x, y) \leftarrow \text{input}_{R}^{bf}(x) \land \text{output}_{T}^{bbf}(x, a, y)$$
$$\text{input}_{T}^{bbf}(x, a) \leftarrow \text{input}_{R}^{bf}(x)$$

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)

\( \leadsto \) 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

Different scenarios, different optimal solutions

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

⇒ 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