DATABASE THEORY

Lecture 4: Complexity of FO Query Answering

Markus Krötzsch

TU Dresden, 21 April 2016
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
How to Measure Query Answering Complexity

Query answering as decision problem
\( \Rightarrow \) consider Boolean queries

Various notions of complexity:
- Combined complexity (complexity w.r.t. size of query and database instance)
- Data complexity (worst case complexity for any fixed query)
- Query complexity (worst case complexity for any fixed database instance)

Various common complexity classes:

\[ L \subseteq \text{NL} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSpace} \subseteq \text{ExpTime} \]
function Eval($\varphi, \mathcal{I}$)

01    \textbf{switch} ($\varphi$) {
02        \textbf{case} $p(c_1, \ldots, c_n)$: \textbf{return} $\langle c_1, \ldots, c_n \rangle \in p^\mathcal{I}$
03        \textbf{case} $\neg \psi$: \textbf{return} $\neg \text{Eval}(\psi, \mathcal{I})$
04        \textbf{case} $\psi_1 \land \psi_2$: \textbf{return} $\text{Eval}(\psi_1, \mathcal{I}) \land \text{Eval}(\psi_2, \mathcal{I})$
05        \textbf{case} $\exists x. \psi$:
06            \textbf{for} $c \in \Delta^\mathcal{I}$ {
07                \textbf{if} \text{Eval}($\psi[x \mapsto c], \mathcal{I}$) \textbf{then} \textbf{return} $\text{true}$
08            }$
09        \textbf{return} \text{false}$
10    }
Let $m$ be the size of $\varphi$, and let $n = |\mathcal{I}|$ (total table sizes)

- How many recursive calls of Eval are there?
  $\leadsto$ one per subexpression: at most $m$

- Maximum depth of recursion?
  $\leadsto$ bounded by total number of calls: at most $m$

- Maximum number of iterations of for loop?
  $\leadsto |\Delta^\mathcal{I}| \leq n$ per recursion level
  $\leadsto$ at most $n^m$ iterations

- Checking $\langle c_1, \ldots, c_n \rangle \in p^\mathcal{I}$ can be done in linear time w.r.t. $n$

Runtime in $m \cdot n^m \cdot n = m \cdot n^{m+1}$
Let $m$ be the size of $\varphi$, and let $n = |\mathcal{I}|$ (total table sizes)

Runtime in $m \cdot n^{m+1}$

Time complexity of FO query evaluation

- Combined complexity: in $\text{ExpTime}$
- Data complexity ($m$ is constant): in $\text{P}$
- Query complexity ($n$ is constant): in $\text{ExpTime}$
We can get better complexity bounds by looking at memory.

Let \( m \) be the size of \( \varphi \), and let \( n = |\mathcal{I}| \) (total table sizes).

- For each (recursive) call, store pointer to current subexpression of \( \varphi \): \( \log m \)
- For each variable in \( \varphi \) (at most \( m \)), store current constant assignment (as a pointer): \( m \cdot \log n \)
- Checking \( \langle c_1, \ldots, c_n \rangle \in p^\mathcal{I} \) can be done in logarithmic space w.r.t. \( n \)

Memory in \( m \log m + m \log n + \log n = m \log m + (m + 1) \log n \)
Let $m$ be the size of $\varphi$, and let $n = |\mathcal{I}|$ (total table sizes)

Memory in $m \log m + (m + 1) \log n$

Space complexity of FO query evaluation

- Combined complexity: in $\text{PSPACE}$
- Data complexity ($m$ is constant): in $\text{L}$
- Query complexity ($n$ is constant): in $\text{PSPACE}$
FO Combined Complexity

The algorithm shows that FO query evaluation is in \( \text{PSPACE} \). Is this the best we can get?

Hardness proof: reduce a known \( \text{PSPACE} \)-hard problem to FO query evaluation
FO Combined Complexity

The algorithm shows that FO query evaluation is in $\text{PSpace}$. Is this the best we can get?

Hardness proof: reduce a known $\text{PSpace}$-hard problem to FO query evaluation

$\leadsto$ QBF satisfiability

Let $Q_1 X_1. Q_2 X_2. \cdots Q_n X_n. \varphi[X_1, \ldots, X_n]$ be a QBF (with $Q_i \in \{\forall, \exists\}$)

- Database instance $I$ with $\Delta^I = \{0, 1\}$
- One table with one row: $\text{true}(1)$
- Transform input QBF into Boolean FO query

$$Q_1 x_1. Q_2 x_2. \cdots Q_n x_n. \varphi[X_1 \mapsto \text{true}(x_1), \ldots, X_n \mapsto \text{true}(x_n)]$$
**PSPACE-hardness for DI Queries**

The previous reduction from QBF may lead to a query that is not domain independent

Example: QBF $\exists p. \neg p$ leads to FO query $\exists x. \neg \text{true}(x)$
PSpace-hardness for DI Queries

The previous reduction from QBF may lead to a query that is not domain independent

Example: QBF $\exists p. \neg p$ leads to FO query $\exists x. \neg true(x)$

Better approach:

- Consider QBF $\Box_1 X_1 \cdot \Box_2 X_2 \cdot \cdots \cdot \Box_n X_n \cdot \varphi[X_1, \ldots, X_n]$ with $\varphi$ in negation normal form: negations only occur directly before variables $X_i$ (still PSpace-complete: exercise)
- Database instance $\mathcal{I}$ with $\Delta^\mathcal{I} = \{0, 1\}$
- Two tables with one row each: true(1) and false(0)
- Transform input QBF into Boolean FO query

$$\Box_1 x_1 \cdot \Box_2 x_2 \cdot \cdots \cdot \Box_n x_n \cdot \varphi'$$

where $\varphi'$ is obtained by replacing each negated variable $\neg X_i$ with false($x_i$) and each non-negated variable $X_i$ with true($x_i$).
The evaluation of FO queries is $\mathsf{PSpace}$-complete with respect to combined complexity.

We have actually shown something stronger:

The evaluation of FO queries is $\mathsf{PSpace}$-complete with respect to query complexity.
The algorithm showed that FO query evaluation is in $L$

~⇒ can we do any better?

What could be better than $L$?

$\mathcal{C} \subseteq L \subseteq \text{NL} \subseteq \text{P} \subseteq \ldots$

~⇒ we need to define circuit complexities first
Definition

A **Boolean circuit** is a finite, directed, acyclic graph where

- each node that has no predecessors is an **input node**
- each node that is not an input node is one of the following types of **logical gate**: AND, OR, NOT
- one or more nodes are designated **output nodes**

\[ \Rightarrow \text{we will only consider Boolean circuits with exactly one output} \]

\[ \Rightarrow \text{propositional logic formulae are Boolean circuits with one output and gates of fanout} \leq 1 \]
Example

A Boolean circuit over an input string $x_1x_2\ldots x_n$ of length $n$

Corresponds to formula \((x_1 \land x_2) \lor (x_1 \land x_3) \lor \ldots \lor (x_{n-1} \land x_n)\)

\(\sim\) accepts all strings with at least two 1s
Circuits as a Model for Parallel Computation

Previous example:

\[ x_1 \cdots x_n \]\n
\( n^2 \) gates

\( \sim n^2 \) processors working in parallel

\( \sim \) computation finishes in 2 steps

- **size**: number of gates = total number of computing steps
- **depth**: longest path of gates = time for parallel computation

\( \sim \) refinement of polynomial time taking parallelizability into account
Solving Problems With Circuits

Observation: the input size is “hard-wired” in circuits

→ each circuit only has a finite number of different inputs
→ not a computationally interesting problem

How can we solve interesting problems with Boolean circuits?
Solving Problems With Circuits

Observation: the input size is “hard-wired” in circuits

→ each circuit only has a finite number of different inputs

→ not a computationally interesting problem

How can we solve interesting problems with Boolean circuits?

**Definition**

A **uniform family** of Boolean circuits is a set of circuits $C_n$ ($n \geq 0$) that can be computed from $n$ (usually in logarithmic space or time; we don’t discuss the details here).

A language $\mathcal{L} \subseteq \{0, 1\}^*$ is **decided by** a uniform family $(C_n)_{n \geq 0}$ of Boolean circuits if for each word $w$ of length $|w|$: 

$$w \in \mathcal{L} \quad \text{if and only if} \quad C_{|w|}(w) = 1$$
Measuring Complexity with Boolean Circuits

How to measure the computing power of Boolean circuits?

Relevant metrics:

- **size** of the circuit: overall number of gates (as function of input size)
- **depth** of the circuit: longest path of gates (as function of input size)
- **fan in**: two inputs per gate or any number of inputs per gate?

Important classes of circuits: small-depth circuits

**Definition**

\((C_n)_{n \geq 0}\) is a family of **small-depth circuits** if

- the size of \(C_n\) is polynomial in \(n\),
- the depth of \(C_n\) is poly-logarithmic in \(n\), that is, \(O(\log^k n)\).
The Complexity Classes $\text{NC}$ and $\text{AC}$

Two important types of small-depth circuits

**Definition**

$\text{NC}^k$ is the class of problems that can be solved by uniform families of circuits $(C_n)_{n \geq 0}$ of fan-in $\leq 2$, size polynomial in $n$, and depth in $O(\log^k n)$.

The class $\text{NC}$ is defined as $\text{NC} = \bigcup_{k \geq 0} \text{NC}^k$.

(“Nick’s Class” named after Nicholas Pippenger by Stephen Cook)

**Definition**

$\text{AC}^k$ and $\text{AC}$ are defined like $\text{NC}^k$ and $\text{NC}$, respectively, but for circuits with arbitrary fan-in.

(A is for “Alternating”: AND-OR gates alternate in such circuits)
family of polynomial size, 
constant depth, 
arbitrary fan-in circuits

\( \sim \) in \( \text{AC}^0 \)
family of polynomial size, constant depth, arbitrary fan-in circuits \( \sim \) in \( \text{AC}^0 \)

We can eliminate arbitrary fan-ins by using more layers of gates:

family of polynomial size, logarithmic depth, bounded fan-in circuits \( \sim \) in \( \text{NC}^1 \)
Relationships of Circuit Complexity Classes

The previous sketch can be generalised:

\[ \text{NC}^0 \subseteq \text{AC}^0 \subseteq \text{NC}^1 \subseteq \text{AC}^1 \subseteq \ldots \subseteq \text{AC}^k \subseteq \text{NC}^{k+1} \subseteq \ldots \]

Only few inclusions are known to be proper: \( \text{NC}^0 \subset \text{AC}^0 \subset \text{NC}^1 \)
Relationships of Circuit Complexity Classes

The previous sketch can be generalised:

\[
\text{NC}^0 \subseteq \text{AC}^0 \subseteq \text{NC}^1 \subseteq \text{AC}^1 \subseteq \ldots \subseteq \text{AC}^k \subseteq \text{NC}^{k+1} \subseteq \ldots
\]

Only few inclusions are known to be proper: \(\text{NC}^0 \subset \text{AC}^0 \subset \text{NC}^1\)

Direct consequence of above hierarchy: \(\text{NC} = \text{AC}\)

Interesting relations to other classes:

\[
\text{NC}^0 \subset \text{AC}^0 \subset \text{NC}^1 \subseteq \text{L} \subseteq \text{NL} \subseteq \text{AC}^1 \subseteq \ldots \subseteq \text{NC} \subseteq \text{P}
\]

Intuition:

- Problems in \(\text{NC}\) are parallelisable
- Problems in \(\text{P} \setminus \text{NC}\) are inherently sequential

However: it is not known if \(\text{NC} \neq \text{P}\)
Theorem

The evaluation of FO queries is complete for (logtime uniform) $\text{AC}^0$ with respect to data complexity.

Proof:

- **Membership:** For a fixed Boolean FO query, provide a uniform construction for a small-depth circuit based on the size of a database

- **Hardness:** Show that circuits can be transformed into Boolean FO queries in logarithmic time (not on a standard TM . . . not in this lecture)
Assumption:
- query and database schema is fixed
- database instance (and thus active domain) are variable

Construct circuit uniformly based on size of active domain
From Query to Circuit

Assumption:
- query and database schema is fixed
- database instance (and thus active domain) are variable

Construct circuit uniformly based on size of active domain

Sketch of construction:
- one input node for each possible database tuple (over given schema and active domain)
  ~ true or false depending on whether tuple is present or not
- Recursively, for each subformula, introduce a gate for each possible tuple (instantiation) of this formula
  ~ true or false depending on whether the subformula holds for this tuple or not
- Logical operators correspond to gate types: basic operators obvious, \( \forall \) as generalised conjunction, \( \exists \) as generalised disjunction
- subformula with \( n \) free variables ~ \(|\text{adom}|^n\) gates
  ~ especially: \(|\text{adom}|^0 = 1\) output gate for Boolean query
Example

We consider the formula

\[
\exists z. (\exists x. \exists y. R(x, y) \land S(y, z)) \land \neg R(a, z)
\]

Over the database instance:

<table>
<thead>
<tr>
<th>R:</th>
<th>S:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(b)</td>
</tr>
<tr>
<td>(a)</td>
<td>(b)</td>
</tr>
<tr>
<td>(a)</td>
<td>(c)</td>
</tr>
</tbody>
</table>

Active domain: \(\{a, b, c\}\)
Example: $\exists z. (\exists x. \exists y. R(x, y) \land S(y, z)) \land \neg R(a, z)$

<table>
<thead>
<tr>
<th></th>
<th>$R(a, a)$</th>
<th>$R(a, b)$</th>
<th>$R(a, c)$</th>
<th>$S(a, a)$</th>
<th>$S(b, a)$</th>
<th>$S(b, b)$</th>
<th>$S(b, c)$</th>
<th>$\ldots$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$\ldots$</td>
<td>0</td>
<td>$\ldots$</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Markus Krötzsch, 21 April 2016

Database Theory

slide 30 of 41
Example: $\exists z. (\exists x. \exists y. R(x, y) \land S(y, z)) \land \neg R(a, z)$
Example: \( \exists z. (\exists x. \exists y. R(x, y) \land S(y, z)) \land \neg R(a, z) \)
Example: $\exists z. (\exists x. \exists y. R(x, y) \land S(y, z)) \land \neg R(a, z)$
Example: \( \exists z. (\exists x. \exists y. R(x, y) \land S(y, z)) \land \neg R(a, z) \)
Example: $\exists z. (\exists x. \exists y. R(x, y) \land S(y, z)) \land \neg R(a, z)$
Example: $\exists z. (\exists x. \exists y. R(x, y) \land S(y, z)) \land \neg R(a, z)$
Example: $\exists z. (\exists x. \exists y. R(x, y) \land S(y, z)) \land \neg R(a, z)$
Example: $\exists z. (\exists x. \exists y. R(x, y) \land S(y, z)) \land \neg R(a, z)$
Example: $\exists z.(\exists x.\exists y. R(x, y) \land S(y, z)) \land \neg R(a, z)$
Example: $\exists z. (\exists x. \exists y. R(x, y) \land S(y, z)) \land \neg R(a, z)$
Summary and Outlook

The evaluation of FO queries is

- $\text{PSPACE}$-complete for combined complexity
- $\text{PSPACE}$-complete for query complexity
- $\text{AC}^0$-complete for data complexity

Circuit complexities help to identify highly parallelisable problems in $\text{P}$

Open questions:

- Which other computing problems are interesting? (next lecture)
- Are there query languages with lower complexities?
- How can we study the expressiveness of query languages?