Overview

1. Introduction | Relational data model
2. First-order queries
3. Complexity of query answering
4. Complexity of FO query answering
5. Query optimization
6. Conjunctive queries
7. Limits of first-order query expressiveness
8. Introduction to Datalog
9. Implementation techniques for Datalog
10. Path queries
11. Constraints (1)
12. Constraints (2)
13. “Buffer time”
14. Outlook: database theory in practice
How to Measure Query Answering Complexity

Query answering as decision problem
⇒ 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:

\[ L \subseteq NL \subseteq P \subseteq NP \subseteq PSpace \subseteq ExpTime \]
function Eval(\(\varphi, I\))

01 \textbf{switch} (\(\varphi\)) {
02 \quad \textbf{case } p(c_1, \ldots, c_n): \text{ return } \langle c_1, \ldots, c_n \rangle \in p^I \n03 \quad \textbf{case } \neg \psi: \text{ return } \neg \text{Eval}(\psi, I) \n04 \quad \textbf{case } \psi_1 \land \psi_2: \text{ return } \text{Eval}(\psi_1, I) \land \text{Eval}(\psi_2, I) \n05 \quad \textbf{case } \exists x. \psi: 
06 \quad \quad \textbf{for } c \in \Delta^I \{ 
07 \quad \quad \quad \textbf{if } \text{Eval}(\psi[x \mapsto c], I) \text{ then return true} \n08 \quad \quad \} 
09 \quad \text{return false} 
10 \}
FO Algorithm Worst-Case Runtime

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 $n = |I|$ (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 $P$
- Query complexity ($n$ is constant): in $\text{ExpTime}$
FO Algorithm Worst-Case Memory Usage

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 $m \log m + m \log n + \log n = m \log m + (m + 1) \log n$
Space Complexity of FO Algorithm

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 $\mathsf{PSpace}$
- Data complexity ($m$ is constant): in $\mathsf{L}$
- Query complexity ($n$ is constant): in $\mathsf{PSpace}$
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

$\sim$ QBF satisfiability

Let $\mathcal{O}_1 x_1 . \mathcal{O}_2 x_2 . \cdots . \mathcal{O}_n x_n . \varphi[X_1, \ldots, X_n]$ be a QBF (with $\mathcal{O}_i \in \{\forall, \exists\}$)

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

$$\mathcal{O}_1 x_1 . \mathcal{O}_2 x_2 . \cdots . \mathcal{O}_n x_n . \varphi[X_1 \mapsto \text{true}(x_1), \ldots, X_n \mapsto \text{true}(x_n)]$$
**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 $\exists x. \neg \text{true}(x)$

Better approach:
- Consider QBF $Q_1X_1.Q_2X_2. \cdots Q_nX_n.\varphi[X_1, \ldots, X_n]$ 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

\[
Q_1x_1.Q_2x_2. \cdots Q_nx_n.\varphi'
\]

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$(x_i)$.
Theorem
The evaluation of FO queries is $\text{PSPACE}$-complete with respect to combined complexity.

We have actually shown something stronger:

Theorem
The evaluation of FO queries is $\text{PSPACE}$-complete with respect to query complexity.
The algorithm showed that FO query evaluation is in \( L \)

\[ \rightsquigarrow \text{can we do any better?} \]

What could be better than \( L \)?

\[ \text{?} \subseteq L \subseteq \text{NL} \subseteq \text{P} \subseteq \ldots \]

\[ \rightsquigarrow \text{we need to define circuit complexities first} \]
Boolean Circuits

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

implies we will only consider Boolean circuits with exactly one output

implies propositional logic formulae are Boolean circuits with one output and gates of fanout \( \leq 1 \)
Example

A Boolean circuit over an input string $x_1 x_2 \ldots x_n$ of length $n$

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

$\leadsto$ accepts all strings with at least two 1s
Circuits as a Model for Parallel Computation

Previous example:

\[ \cdots\]

\[
\begin{array}{cccccc}
\Diamond & \Diamond & \Diamond & \Diamond & \Diamond \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
\cdots & \cdots & \cdots & \cdots & \cdots \\
\]

\[ n^2 \text{ processors working in parallel} \]

\[ \sim \text{computation finishes in 2 steps} \]

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

\[ \sim \text{refinement of polynomial time taking parallelizability into account} \]
Observation: the input size is “hard-wired” in circuits
\(\leadsto\) each circuit only has a finite number of different inputs
\(\leadsto\) not a computationally interesting problem

How can we solve interesting problems with Boolean circuits?
Observation: the input size is “hard-wired” in circuits
\[ \rightarrow \text{each circuit only has a finite number of different inputs} \]
\[ \rightarrow \text{not a computationally interesting problem} \]

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 \(|w|\):

\[ w \in \mathcal{L} \quad \text{if and only if} \quad C_{|w|}(w) = 1 \]
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)
Example

family of polynomial size, constant depth, arbitrary fan-in circuits
$\leadsto$ in $AC^0$

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

family of polynomial size, logarithmic depth, bounded fan-in circuits
$\leadsto$ in $NC^1$
Relationships of Circuit Complexity Classes

The previous sketch can be generalised:

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

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

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

Interesting relations to other classes:

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

Intuition:

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

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

The evaluation of FO queries is complete for (logtime uniform) \( AC^0 \) 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 \ldots not in this lecture).
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)
  \(\sim\) 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
  \(\sim\) 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 \(\sim\) \(|\text{adom}|^n\) gates
  \(\sim\) 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>b</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>

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