DATABASE THEORY

Lecture 3: Complexity of Query Answering

Markus Krötzsch
Knowledge-Based Systems

TU Dresden, 17th Apr 2018
Review: The Relational Calculus

What we have learned so far:

- There are many ways to describe databases:
  - \( \sim \) named perspective, unnamed perspective, interpretations, ground fracts, (hyper)graphs

- There are many ways to describe query languages:
  - \( \sim \) relational algebra, domain independent FO queries, safe-range FO queries, active domain FO queries, Codd’s tuple calculus
  - \( \sim \) either under named or under unnamed perspective

All of these are largely equivalent: The Relational Calculus

Next question: How hard is it to answer such queries?
How to Measure Complexity of Queries?

- Complexity classes often for decision problems (yes/no answer)
  \(\leadsto\) database queries return many results (no decision problem)

- The size of a query result can be very large
  \(\leadsto\) it would not be fair to measure this as “complexity”

- In practice, database instances are much larger than queries
  \(\leadsto\) can we take this into account?
We consider the following decision problems:

- **Boolean query entailment:** given a Boolean query $q$ and a database instance $I$, does $I \models q$ hold?

- **Query of tuple problem:** given an $n$-ary query $q$, a database instance $I$ and a tuple $\langle c_1, \ldots, c_n \rangle$, does $\langle c_1, \ldots, c_n \rangle \in M[q](I)$ hold?

- **Query emptiness problem:** given a query $q$ and a database instance $I$, does $M[q](I) \neq \emptyset$ hold?

$\leadsto$ Computationally equivalent problems (exercise)
The Size of the Input

**Combined Complexity**
Input: Boolean query $q$ and database instance $\mathcal{I}$
Output: Does $\mathcal{I} \models q$ hold?

$\sim$ estimates complexity in terms of overall input size
$\sim$ “2KB query/2TB database” = “2TB query/2KB database”
The Size of the Input

**Combined Complexity**
Input: Boolean query $q$ and database instance $\mathcal{I}$
Output: Does $\mathcal{I} \models q$ hold?

→ estimates complexity in terms of overall input size
→ “2KB query/2TB database” = “2TB query/2KB database”
→ study worst-case complexity of algorithms for fixed queries:

**Data Complexity**
Input: database instance $\mathcal{I}$
Output: Does $\mathcal{I} \models q$ hold? (for fixed $q$)
The Size of the Input

**Combined Complexity**
Input: Boolean query $q$ and database instance $I$
Output: Does $I \models q$ hold?

→ estimates complexity in terms of overall input size
→ “2KB query/2TB database” = “2TB query/2KB database”
→ study worst-case complexity of algorithms for fixed queries:

**Data Complexity**
Input: database instance $I$
Output: Does $I \models q$ hold? (for fixed $q$)

→ we can also fix the database and vary the query:

**Query Complexity**
Input: Boolean query $q$
Output: Does $I \models q$ hold? (for fixed $I$)
Review: Computation and Complexity Theory
Computation is usually modelled with Turing Machines (TMs)
\[\sim \text{“algorithm” = “something implemented on a TM”}\]

A TM is an automaton with (unlimited) working memory:
- It has a finite set of states \( Q \)
- \( Q \) includes a start state \( q_{\text{start}} \) and an accept state \( q_{\text{acc}} \)
- The memory is a tape with numbered cells 0, 1, 2, \ldots
- Each tape cell holds one symbol from the set of tape symbols \( \Gamma \)
- There is a special symbol \( \_ \) for empty tape cells
- The TM has a transition relation \( \Delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{l, r, s\}) \)
- \( \Delta \) might be a partial function \( (Q \times \Gamma) \rightarrow (Q \times \Gamma \times \{l, r, s\}) \)

There are many different but equivalent ways of defining TMs.
The Turing Machine (2)

TMs operate step-by-step:

- At every moment, the TM is in one state \( q \in Q \) with its read/write head at a certain tape position \( p \in \mathbb{N} \), and the tape has a certain contents \( \sigma_0 \sigma_1 \sigma_2 \cdots \) with all \( \sigma_i \in \Gamma \)
  \( \leadsto \) current configuration of the TM
- The TM starts in state \( q_{\text{start}} \) and at tape position 0.
- Transition \( \langle q, \sigma, q', \sigma', d \rangle \in \Delta \) means:
  if in state \( q \) and the tape symbol at its current position is \( \sigma \),
  then change to state \( q' \), write symbol \( \sigma' \) to tape, move head by \( d \) (left/right/stay)
- If there is more than one possible transition, the TM picks one nondeterministically
- The TM halts when there is no possible transition for the current configuration (possibly never)

A computation path (or run) of a TM is a sequence of configurations that can be obtained by some choice of transition.
Languages Accepted by TMs

The (nondeterministic) TM accepts an input $\sigma_1 \cdots \sigma_n \in (\Gamma \setminus \{\omega\})^*$ if, when started on the tape $\sigma_1 \cdots \sigma_n \omega \omega \cdots$, 
(1) the TM halts on every computation path and 
(2) there is at least one computation path that halts in the accepting state $q_{\text{acc}} \in Q$. 

accept: 

reject: 

reject (not halting): 

Markus Krötzsch, 17th Apr 2018
A decision problem is a language \( \mathcal{L} \) of words over \( \Sigma = \Gamma \setminus \{\sqcup\} \)
\( \leadsto \) the set of all inputs for which the answer is “yes”

A TM decides a decision problem \( \mathcal{L} \) if it accepts exactly the words in \( \mathcal{L} \)

TMs take time (number of steps) and space (number of cells):

- Time\( (f(n)) \): Problems that can be decided by a DTM in \( O(f(n)) \) steps, where \( f \) is a function of the input length \( n \)
- Space\( (f(n)) \): Problems that can be decided by a DTM using \( O(f(n)) \) tape cells, where \( f \) is a function of the input length \( n \)
A decision problem is a language $L$ of words over $\Sigma = \Gamma \setminus \{\downarrow\}$

$\leadsto$ the set of all inputs for which the answer is “yes”

A TM decides a decision problem $L$ if it accepts exactly the words in $L$

TMs take time (number of steps) and space (number of cells):

- Time($f(n)$): Problems that can be decided by a DTM in $O(f(n))$ steps, where $f$ is a function of the input length $n$
- Space($f(n)$): Problems that can be decided by a DTM using $O(f(n))$ tape cells, where $f$ is a function of the input length $n$
- NTime($f(n)$): Problems that can be decided by a TM in at most $O(f(n))$ steps on any of its computation paths
- NSpace($f(n)$): Problems that can be decided by a TM using at most $O(f(n))$ tape cells on any of its computation paths
Some Common Complexity Classes

\[
P = \text{PTime} = \bigcup_{k \geq 1} \text{Time}(n^k)
\]

\[
\text{Exp} = \text{ExpTime} = \bigcup_{k \geq 1} \text{Time}(2^n^k)
\]

\[
2\text{Exp} = 2\text{ExpTime} = \bigcup_{k \geq 1} \text{Time}(2^{2^n^k})
\]

\[
\text{ETime} = \bigcup_{k \geq 1} \text{Time}(2^{n^k})
\]

\[
L = \text{LogSpace} = \text{Space}(\log n)
\]

\[
\text{PSpace} = \bigcup_{k \geq 1} \text{Space}(n^k)
\]

\[
\text{ExpSpace} = \bigcup_{k \geq 1} \text{Space}(2^n^k)
\]

\[
\text{NP} = \bigcup_{k \geq 1} \text{NTime}(n^k)
\]

\[
\text{NExp} = \text{NExpTime} = \bigcup_{k \geq 1} \text{NTime}(2^n^k)
\]

\[
\text{N2Exp} = \text{N2ExpTime} = \bigcup_{k \geq 1} \text{NTime}(2^{2^n^k})
\]

\[
\text{NL} = \text{NLogSpace} = \text{NSpace}(\log n)
\]

\[
\text{ExpSpace} = \bigcup_{k \geq 1} \text{Space}(2^n^k)
\]
NP

NP = Problems for which a possible solution can be verified in P:

- for every $w \in L$, there is a certificate $c_w \in \Sigma^*$, such that
- the length of $c_w$ is polynomial in the length of $w$, and
- the language $\{w##c_w \mid w \in L\}$ is in P

Equivalent to definition with nondeterministic TMs:

- $\Rightarrow$ nondeterministically guess certificate; then run verifier DTM
- $\Leftarrow$ use accepting polynomial run as certificate; verify TM steps
NP Examples

Examples:

- Sudoku solvability (certificate: filled-out grid)
- Composite (non-prime) number (certificate: factorization)
- Prime number (certificate: see Wikipedia “Primality certificate”)
- Propositional logic satisfiability (certificate: satisfying assignment)
- Graph colourability (certificate: coloured graph)
NP and coNP

Note: Definition of NP is not symmetric

- there does not seem to be any polynomial certificate for Sudoku unsolvability or logic unsatisfiability
- converse of an NP problem is coNP
- similar for NExpTime and N2ExpTime

Other classes are symmetric:

- Deterministic classes (coP = P etc.)
- Space classes mentioned above (esp. coNL = NL)
Reductions

Observation: some problems can be reduced to others

Example: 3-colouring can be reduced to propositional satisfiability

Encoding colours in propositions:
• $r_i$ means "vertex $i$ is red"
• $g_i$ means "vertex $i$ is green"
• $b_i$ means "vertex $i$ is blue"

Colouring conditions on vertices:
\[(r_1 \land \neg g_1 \land \neg b_1) \lor (\neg r_1 \land g_1 \land \neg b_1) \lor (\neg r_1 \land \neg g_1 \land b_1)\] (and so on for all vertices)

Colouring conditions for edges:
\[\neg (r_1 \land r_2) \land \neg (g_1 \land g_2) \land \neg (b_1 \land b_2)\] (and so on for all edges)

Satisfying truth assignment $\iff$ valid colouring
Reductions

Observation: some problems can be reduced to others

Example: 3-colouring can be reduced to propositional satisfiability

![Diagram of a graph with vertices and edges labeled for 3-coloring]

Encoding colours in propositions:

- $r_i$ means "vertex $i$ is red"
- $g_i$ means "vertex $i$ is green"
- $b_i$ means "vertex $i$ is blue"

Colouring conditions on vertices:

\[(r_1 \land \neg g_1 \land \neg b_1) \lor (\neg r_1 \land g_1 \land \neg b_1) \lor (\neg r_1 \land \neg g_1 \land b_1)\]

(and so on for all vertices)

Colouring conditions for edges:

\[\neg (r_1 \land r_2) \land \neg (g_1 \land g_2) \land \neg (b_1 \land b_2)\]

(and so on for all edges)

Satisfying truth assignment $\iff$ valid colouring
Reductions

Observation: some problems can be reduced to others

Example: 3-colouring can be reduced to propositional satisfiability

Encoding colours in propositions:

- \( r_i \) means "vertex \( i \) is red"
- \( g_i \) means "vertex \( i \) is green"
- \( b_i \) means "vertex \( i \) is blue"

Colouring conditions on vertices:

\[
(r_1 \land \neg g_1 \land \neg b_1) \lor (\neg r_1 \land g_1 \land \neg b_1) \lor (\neg r_1 \land \neg g_1 \land b_1) \lor \ldots
\]

Colouring conditions for edges:

\[
\neg (r_1 \land r_2) \land \neg (g_1 \land g_2) \land \neg (b_1 \land b_2) \lor \ldots
\]
Reductions

Observation: some problems can be reduced to others

Example: 3-colouring can be reduced to propositional satisfiability

Encoding colours in propositions:
- $r_i$ means "vertex $i$ is red"
- $g_i$ means "vertex $i$ is green"
- $b_i$ means "vertex $i$ is blue"

Colouring conditions on vertices: $(r_1 \land \neg g_1 \land \neg b_1) \lor (\neg r_1 \land g_1 \land \neg b_1) \lor (\neg r_1 \land \neg g_1 \land b_1)$ (and so on for all vertices)

Colouring conditions for edges:
- $\neg(r_1 \land r_2) \land \neg(g_1 \land g_2) \land \neg(b_1 \land b_2)$ (and so on for all edges)
Reductions

Observation: some problems can be reduced to others

Example: 3-colouring can be reduced to propositional satisfiability

Encoding colours in propositions:

- \( r_i \) means "vertex \( i \) is red"
- \( g_i \) means "vertex \( i \) is green"
- \( b_i \) means "vertex \( i \) is blue"

Colouring conditions on vertices:

\[
(r_1 \land \neg g_1 \land \neg b_1) \lor (\neg r_1 \land g_1 \land \neg b_1) \lor (\neg r_1 \land \neg g_1 \land b_1)
\]

(and so on for all vertices)

Colouring conditions for edges:

\[

\neg (r_1 \land r_2) \land \neg (g_1 \land g_2) \land \neg (b_1 \land b_2)
\]

(and so on for all edges)

Satisfying truth assignment ⇔ valid colouring
Reductions

Observation: some problems can be reduced to others

Example: 3-colouring can be reduced to propositional satisfiability

Encoding colours in propositions:

- $r_i$ means "vertex $i$ is red"
- $g_i$ means "vertex $i$ is green"
- $b_i$ means "vertex $i$ is blue"

Colouring conditions on vertices:

$$(r_1 \land \neg g_1 \land \neg b_1) \lor (\neg r_1 \land g_1 \land \neg b_1) \lor (\neg r_1 \land \neg g_1 \land b_1) \quad \text{(and so on for all vertices)}$$

Colouring conditions for edges:

$$\neg (r_1 \land r_2) \land \neg (g_1 \land g_2) \land \neg (b_1 \land b_2) \quad \text{(and so on for all edges)}$$

Satisfying truth assignment $\Leftrightarrow$ valid colouring
Defining Reductions

**Definition 3.1:** Consider languages $L_1, L_2 \subseteq \Sigma^*$. A computable function $f : \Sigma^* \rightarrow \Sigma^*$ is a many-one reduction from $L_1$ to $L_2$ if:

$$w \in L_1 \text{ if and only if } f(w) \in L_2$$

- we can solve problem $L_1$ by reducing it to problem $L_2$
- only useful if the reduction is much easier than solving $L_1$ directly
- polynomial many-one reductions
The Structure of NP

Idea: polynomial many-one reductions define an order on problems
The Structure of NP

Idea: polynomial many-one reductions define an order on problems
The Structure of NP

Idea: polynomial many-one reductions define an order on problems
The Structure of NP

Idea: polynomial many-one reductions define an order on problems
The Structure of NP

Idea: polynomial many-one reductions define an order on problems
Theorem 3.2 (Cook 1971; Levin 1973): All problems in NP can be polynomi-
ally many-one reduced to the propositional satisfiability problem (SAT).

- NP has a maximal class that contains a practically relevant problem
- If SAT can be solved in P, all problems in NP can
- Karp discovered 21 further such problems shortly after (1972)
- Thousands such problems have been discovered since . . .
Theorem 3.2 (Cook 1971; Levin 1973): All problems in NP can be polynomially many-one reduced to the propositional satisfiability problem (SAT).

- NP has a maximal class that contains a practically relevant problem
- If SAT can be solved in P, all problems in NP can
- Karp discovered 21 further such problems shortly after (1972)
- Thousands such problems have been discovered since . . .

Definition 3.3: A language is
- NP-hard if every language in NP is polynomially many-one reducible to it
- NP-complete if it is NP-hard and in NP
Comparing Complexity Classes

Is any NP-complete problem in P?

- If yes, then P = NP
- Nobody knows \( \sim \) biggest open problem in computer science
- Similar situations for many complexity classes

Some things that are known:

\[
L \subseteq NL \subseteq P \subseteq NP \subseteq PSpace \subseteq ExpTime \subseteq NExpTime
\]

- None of these is known to be strict
- But we know that P \( \subset \) ExpTime and NL \( \subset \) PSpace
- Moreover PSpace = NPSpace (by Savitch's Theorem)

(see TU Dresden course complexity theory for many more details)
Comparing Complexity Classes

Is any NP-complete problem in P?

- If yes, then $P = NP$
- Nobody knows $\sim$ biggest open problem in computer science
- Similar situations for many complexity classes

Some things that are known:

$L \subseteq NL \subseteq P \subseteq NP \subseteq PSpace \subseteq ExpTime \subseteq NExpTime$

- None of these is known to be strict
- But we know that $P \not\subseteq ExpTime$ and $NL \not\subseteq PSpace$
- Moreover $PSpace = NPSpace$ (by Savitch's Theorem)

(see TU Dresden course complexity theory for many more details)
Comparing Tractable Problems

Polynomial-time many-one reductions work well for (presumably) super-polynomial problems \( \sim \) what to use for P and below?
Comparing Tractable Problems

Polynomial-time many-one reductions work well for (presumably) super-polynomial problems $\sim$ what to use for $P$ and below?

**Definition 3.4:** A LogSpace transducer is a deterministic TM with three tapes:

- a read-only input tape
- a read/write working tape of size $O(\log n)$
- a write-only, write-once output tape

Such a TM needs a slightly different form of transitions:

- transition function input: state, input tape symbol, working tape symbol
- transition function output: state, working tape write symbol, input tape move, working tape move, output tape symbol or $\bot$ to not write anything to the output
The Power of LogSpace

LogSpace transducers can still do a few things:

- store a constant number of counters and increment/decrement the counters
- store a constant number of pointers to the input tape, and locate/read items that start at this address from the input tape
- access/process/compare items from the input tape bit by bit

Example 3.5: Adding and subtracting binary numbers, detecting palindromes, comparing lists, searching items in a list, sorting lists, . . . can all be done in L.
Joining Two Tables in LogSpace

**Input:** two relations $R$ and $S$, represented as a list of tuples

- Use two pointers $p_R$ and $p_S$ pointing to tuples in $R$ and $S$, respectively
- Outer loop: iterate $p_R$ over all tuples of $R$
- Inner loop for each position of $p_R$: iterate $p_S$ over all tuples of $S$
- For each combination of $p_R$ and $p_S$, compare the tuples:
  - Use another two loops that iterate over the columns of $R$ and $S$
  - Compare attribute names bit by bit
  - For matching attribute names, compare the respective tuple values bit by bit
- If all joined columns agree, copy the relevant parts of tuples $p_R$ and $p_S$ to the output (bit by bit)

**Output:** $R \bowtie S$
Joining Two Tables in LogSpace

**Input:** two relations $R$ and $S$, represented as a list of tuples

- Use two pointers $p_R$ and $p_S$ pointing to tuples in $R$ and $S$, respectively
- Outer loop: iterate $p_R$ over all tuples of $R$
- Inner loop for each position of $p_R$: iterate $p_S$ over all tuples of $S$
- For each combination of $p_R$ and $p_S$, compare the tuples:
  - Use another two loops that iterate over the columns of $R$ and $S$
  - Compare attribute names bit by bit
  - For matching attribute names, compare the respective tuple values bit by bit
- If all joined columns agree, copy the relevant parts of tuples $p_R$ and $p_S$ to the output (bit by bit)

**Output:** $R \bowtie S$

$\sim$ Fixed number of pointers and counters
(making this fully formal is still a bit of work; e.g., an additional counter is needed to move the input read head to the target of a pointer (seek))
LogSpace functions: The output of a LogSpace transducer is the contents of its output tape when it halts $\Rightarrow$ a partial function $\Sigma^* \rightarrow \Sigma^*$

Note: the composition of two LogSpace functions is LogSpace (exercise)

**Definition 3.6:** A many-one reduction $f$ from $L_1$ to $L_2$ is a LogSpace reduction if it is implemented by some LogSpace transducer.

$\Rightarrow$ can be used to define hardness for classes P and NL
From L to NL

NL: Problems whose solution can be verified in L

Example: Reachability

- Input: a directed graph $G$ and two nodes $s$ and $t$ of $G$
- Output: accept if there is a directed path from $s$ to $t$ in $G$

Algorithm sketch:

- Store the id of the current node and a counter for the path length
- Start with $s$ as current node
- In each step, increment the counter and move from the current node to one of its direct successors (nondeterministic)
- When reaching $t$, accept
- When the step counter is larger than the total number of nodes, reject
Propositional satisfiability can be solved in linear space:
\[ \sim \text{iterate over possible truth assignments and check each in turn} \]

More generally: all problems in NP can be solved in PSpace
\[ \sim \text{try all conceivable polynomial certificates and verify each in turn} \]

What is a “typical” (that is, hard) problem in PSpace?
\[ \sim \text{Simple two-player games, and other uses of alternating quantifiers} \]
Example: Playing “Geography”

A children’s game:

- Two players are taking turns naming cities.
- Each city must start with the last letter of the previous.
- Repetitions are not allowed.
- The first player who cannot name a new city looses.
Example: Playing “Geography”

A children’s game:

- Two players are taking turns naming cities.
- Each city must start with the last letter of the previous.
- Repetitions are not allowed.
- The first player who cannot name a new city looses.

A mathematicians’ game:

- Two players are marking nodes on a directed graph.
- Each node must be a successor of the previous one.
- Repetitions are not allowed.
- The first player who cannot mark a new node looses.
Example: Playing “Geography”

A children’s game:

- Two players are taking turns naming cities.
- Each city must start with the last letter of the previous.
- Repetitions are not allowed.
- The first player who cannot name a new city looses.

A mathematicians’ game:

- Two players are marking nodes on a directed graph.
- Each node must be a successor of the previous one.
- Repetitions are not allowed.
- The first player who cannot mark a new node looses.

**Question**: given a certain graph and start node, can Player 1 enforce a win (i.e., does he have a winning strategy)?

~ PSpace-complete problem
Example: Quantified Boolean Formulae (QBF)

We consider formulae of the following form:

$$Q_1X_1. Q_2X_2. \cdots Q_nX_n. \varphi[X_1, \ldots, X_n]$$

where $Q_i \in \{\exists, \forall\}$ are quantifiers, $X_i$ are propositional logic variables, and $\varphi$ is a propositional logic formula with variables $X_1, \ldots, X_n$ and constants $\top$ (true) and $\bot$ (false)

Semantics:

- Propositional formulae without variables (only constants $\top$ and $\bot$) are evaluated as usual
- $\exists X_1. \varphi[X_1]$ is true if either $\varphi[X_1/\top]$ or $\varphi[X_1/\bot]$ are
- $\forall X_1. \varphi[X_1]$ is true if both $\varphi[X_1/\top]$ and $\varphi[X_1/\bot]$ are
Example: Quantified Boolean Formulae (QBF)

We consider formulae of the following form:

\[ \mathcal{Q}_1 X_1 . \mathcal{Q}_2 X_2 . \cdots \mathcal{Q}_n X_n . \varphi[X_1, \ldots, X_n] \]

where \( \mathcal{Q}_i \in \{\exists, \forall\} \) are quantifiers, \( X_i \) are propositional logic variables, and \( \varphi \) is a propositional logic formula with variables \( X_1, \ldots, X_n \) and constants \( \top \) (true) and \( \bot \) (false).

Semantics:

- Propositional formulae without variables (only constants \( \top \) and \( \bot \)) are evaluated as usual.
- \( \exists X_1. \varphi[X_1] \) is true if either \( \varphi[X_1/\top] \) or \( \varphi[X_1/\bot] \) are.
- \( \forall X_1. \varphi[X_1] \) is true if both \( \varphi[X_1/\top] \) and \( \varphi[X_1/\bot] \) are.

Question: Is a given QBF formula true?

\( \leadsto \) PSpace-complete problem
A Note on Space and Time

How many different configurations does a TM have in space \((f(n))\)?

\[ |Q| \cdot f(n) \cdot |\Gamma|^f(n) \]

\(\rightarrow\) No halting run can be longer than this

\(\rightarrow\) A time-bounded TM can explore all configurations in time proportional to this
A Note on Space and Time

How many different configurations does a TM have in space \((f(n))\)?

\[ |Q| \cdot f(n) \cdot |\Gamma|^{f(n)} \]

\(\sim\) No halting run can be longer than this

\(\sim\) A time-bounded TM can explore all configurations in time proportional to this

Applications:

- \(L \subseteq P\)
- \(P\text{Space} \subseteq \text{ExpTime}\)
Summary and Outlook

The complexity of query languages can be measured in different ways.

Relevant complexity classes are based on restricting space and time:

\[ L \subseteq NL \subseteq P \subseteq NP \subseteq PSpace \subseteq \text{ExpTime} \]

Problems are compared using many-one reductions.

\[ \rightsquigarrow \text{see TU Dresden course Complexity Theory for further details and deeper insights} \]

Open questions:

- Now how hard is it to answer FO queries? (next lecture)
- We saw that joins are in LogSpace – is this tight?
- How can we study the expressiveness of query languages?