Lecture 5: Conjunctive queries

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. Conjunctive queries (2)
7. Query optimization
8. Limits of first-order query expressiveness
9. Introduction to Datalog
10. Implementation techniques for Datalog
11. Path queries
12. Constraints (1)
13. Constraints (2)
14. Outlook: database theory in practice

Review: FO Query Complexity

The evaluation of FO queries is

- PSpace-complete for combined complexity
- PSpace-complete for query complexity
- AC⁰-complete for data complexity

PsSpace is rather high

conjunctive queries are simpler than that?

Conjunctive Queries

Idea: restrict FO queries to conjunctive, positive features

Definition

A conjunctive query (CQ) is an expression of the form

$$\exists y_1, \ldots, y_m.A_1 \land \ldots \land A_f$$

where each Aᵢ is an atom of the form R(t₁, ..., tₖ). In other words, a conjunctive query is an FO query that only uses conjunctions of atoms and (outer) existential quantifiers.

Example: “Find all lines that depart from an accessible stop” (as seen in earlier lectures)

$$\exists y_{SID}, y_{Stop}, y_{To}.\text{Stops}(y_{SID}, y_{Stop}, "true") \land \text{Connect}(y_{SID}, y_{To}, x_{Line})$$
Conjunctive Queries in Relational Calculus

The expressive power of CQs can also be captured in the relational calculus.

**Definition**

A conjunctive query (CQ) is a relational algebra expression that uses only the operations select $\sigma_{n=m}$, project $\pi_{a_1,...,a_n}$, join $\Join$, and renaming $\delta_{a_1,...,a_n\rightarrow b_1,...,b_n}$.

Renaming is only relevant in named perspective

$\leadsto$ CQs are also known as SELECT-PROJECT-JOIN queries.

Extensions of Conjunctive Queries

Two features are often added:

- **Equality**: CQs with equality can use atoms of the form $t_1 \approx t_2$ (in relational calculus: table constants)
- **Unions**: unions of conjunctive queries are called UCQs (in this case the union is only allowed as outermost operator)

Both extensions truly increase expressive power (as shown in exercise)

Features omitted on purpose: negation and universal quantifiers

$\leadsto$ the reason for this is query complexity (as we shall see)

Boolean Conjunctive Queries

A Boolean conjunctive query (BCQ) asks for a mapping from query variables to domain elements such that all atoms are true.

Example: “Is there an accessible stop where some line departs?”

$\exists y\text{SID}, y\text{Stop}, y\text{To}, y\text{Line}$.\text{Stops}(y\text{SID}, y\text{Stop}, "true") \land \text{Connect}(y\text{SID}, y\text{To}, y\text{Line})$

Stops:

<table>
<thead>
<tr>
<th>SID</th>
<th>Stop</th>
<th>Accessible</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>Hauptbahnof</td>
<td>true</td>
</tr>
<tr>
<td>42</td>
<td>Helmholtzstr.</td>
<td>true</td>
</tr>
<tr>
<td>57</td>
<td>Stadtgutstr.</td>
<td>true</td>
</tr>
<tr>
<td>123</td>
<td>Gustav-Freytag-Str.</td>
<td>false</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Connect:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>57</td>
<td>42</td>
<td>85</td>
</tr>
<tr>
<td>17</td>
<td>789</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

How Hard is it to Answer CQs?

If we know the variable mappings, it is easy to check:

- Checking if a single ground atom $R(c_1, \ldots, c_k)$ holds can be done in linear time
- Checking if a conjunction of ground atoms holds can be done in quadratic time

$\leadsto$ A candidate BCQ match can be verified in $P$

(There are $n^m$ candidates: $n$ size of domain; $m$ number of query variables)

**Theorem**

BCQ query answering is in $NP$ for combined complexity (and also for query complexity).

$\leadsto$ Better than $PSpace$ (presumably)
Can we do any better?

Not really. To see this, let’s look at some other problems.

Consider two relational structures $I$ and $J$ (= database instances, interpretations, hypergraphs)

**Definition**

A homomorphism $h$ from $I$ to $J$ is a function $h : \Delta^I \rightarrow \Delta^J$ such that, for all relation names $R$:

$$\text{if } \langle d_1, \ldots, d_n \rangle \in R^I \text{ then } \langle h(d_1), \ldots, h(d_n) \rangle \in R^J.$$ 

The homomorphism problem is the question if there is a homomorphism from $I$ to $J$.

**Example: Three-colouring as Homomorphism**

$I$:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

$J$:

3-colouring is $NP$-hard

$\Rightarrow$ the homomorphism problem is $NP$-hard

**BCQ Answering as Homomorphism Problem**

The homomorphism problem can be reduced to BCQ answering:

- A relational structure $I$ gives rise to a CQ $Q_I$:
  - replace domain elements by variables (one-to-one); add one query atom per relational tuple; existentially quantify all variables
- $I$ has a homomorphism to $J$ if and only if $J \models Q_I$

BCQ answering can be reduced to the homomorphism problem:

- Clear for BCQs that don’t contain constants
- Eliminate query constants $a$: create new relation $R_a = \{ \langle a \rangle \}$; replace $a$ by a fresh variable $x$ and add a query atom $R_a(x)

$\Rightarrow$ both problems are equivalent

**Complexity of Conjunctive Query Answering**

We showed that BCQ answering is in $NP$ and that the homomorphism problem is $NP$-hard, therefore:

**Theorem**

BCQ answering is

- $NP$-complete for combined complexity
- $NP$-complete for query complexity
- in $AC^0$ for data complexity (inherited from FO queries)
Constraint Satisfaction Problems

Another important problem equivalent to BCQ answering

Definition

A constraint satisfaction problem (CSP) over a domain $\Delta$ is given by a set of variables $\{x_1, \ldots, x_n\}$ and a set of constraints $\{C_1, \ldots, C_m\}$, where each constraint $C_i$ has the form $(X_i, R_i)$ with

- $X_i$ a list of variables from $\{x_1, \ldots, x_n\}$,
- $R_i$ a $|X_i|$-ary relation over $\Delta$.

A solution to the CSP is an assignment of variables to values from $\Delta$ such that all constraints are satisfied (=all tuples occur in the respective relations).

$\leadsto$ alternative notation for BCQ answering/homomorphism problem

CSP Example

A combinatorial crossword puzzle:

Domain: $\Delta = \{A, \ldots, Z\}$
Variables: $x_1, \ldots, x_{26}$
Constraints:

1 vertically: $\text{H E A R T}$
1 horizontally: $\text{H O N E Y}$
3 vertically: $\text{R A D I O}$
3 horizontally: $\text{H A P P Y}$
5 vertically: $\text{B A C H}$

Equivalent Problems

Summing up, the following problems are equivalent:

- Answering a conjunctive query over a database instance
- Finding a homomorphism from a relational structure to another
- Solving a constraint satisfaction problem

Each of these problems is $\text{NP}$-complete

Towards Better Complexities

$\text{NP}$-complete problems are still intractable

$\leadsto$ can we do better?

Problem: searching a match may require backtracking, eventually exploring all options

Intuition: life would be easier if we would not have to go back so much . . .

$\leadsto$ the problem is with the cycles
Example: Cyclic CQs

"Is there a child whose parents are married with each other?"

\[ \exists y_c, y_m, y_f. \text{mother}(y_c, y_m) \land \text{father}(y_c, y_f) \land \text{married}(y_m, y_f) \]

\[ \leadsto \text{cyclic query} \]

Example: Acyclic CQs

"Is there a child whose parents are married with someone?"

\[ \exists y_c, y_m, y_f, y_{mm}, y_{mf}. \text{mother}(y_c, y_m) \land \text{father}(y_c, y_f) \land \text{married}(y_m, y_{mm}) \land \text{married}(y_{mf}, y_f) \]

\[ \leadsto \text{acyclic query} \]

Defining Acyclic Queries

Queries in general are hypergraphs

\[ \leadsto \text{What does “acyclic” mean?} \]

View hypergraphs as graphs to check acyclicity?

- **Primal graph**: same vertices; edges between each pair of vertices that occur together in a hyperedge
- **Incidence graph**: vertices and hyperedges as vertices, with edges to mark incidence (bipartite graph)

However: both graphs have cycles in almost all cases

Acyclic Hypergraphs

**GYO-reduction** algorithm to check acyclicity:

(after Graham [1979] and Yu & Özsoyolu [1979])

Input: hypergraph \( H = \langle V, E \rangle \) (we don’t need relation labels here)

Output: GYO-reduct of \( H \)

Apply the following simplification rules as long as possible:

1. Delete all vertices that occur in at most one hyperedge
2. Delete all hyperedges that are empty or that are contained in other hyperedges

**Definition**

A hypergraph is **acyclic** if its GYO-reduct is \( \langle \emptyset, \emptyset \rangle \).

A CQ is **acyclic** if its associated hypergraph is.
Example 1: GYO-Reduction

Example 2: GYO-Reduction

Alternative Version of GYO-Reduction

An ear of a hypergraph \( \langle V, E \rangle \) is a hyperedge \( e \in E \) that satisfies one of the following:

1. there is an edge \( e' \in E \) such that \( e \neq e' \) and every vertex of \( e \) is either only in \( e \) or also in \( e' \), or
2. \( e \) has no intersection with any other hyperedge.

Example:

edges \( \langle 4, 5, 6 \rangle \) and \( \langle 7, 8, 9 \rangle \) are ears
GYO'-Reduction

Input: hypergraph $H = \langle V, E \rangle$
Output: GYO'-reduct of $H$

Apply the following simplification rule as long as possible:

- Select an ear $e$ of $H$
- Delete $e$
- Delete all vertices that only occurred in $e$

Theorem

The GYO-reduct is $\langle \emptyset, \emptyset \rangle$ if and only if the GYO'-reduct is $\langle \emptyset, \emptyset \rangle$

$\leadsto$ alternative characterization of acyclic hypergraphs

--

Join Trees

Both GYO algorithms can be implemented in linear time

Open question: what benefit does BCQ acyclicity give us?

Fact: if a BCQ is acyclic, then it has a join tree

Definition

A join tree of a (B)CQ is an arrangement of its query atoms in a tree structure $T$, such that for each variable $x$, the atoms that refer to $x$ are a connected subtree of $T$.

A (B)CQ that has a join tree is called a tree query.

Example: Join Tree

$\exists x, y, z, t, u, v, w. (r(x, y, z) \land r(t, u, y) \land s(u, v, y, z) \land q(t, w))$

Processing Join Trees Efficiently

Join trees can be processed in polynomial time

Key ingredient: the semijoin operation

Definition

Given two relations $R[U]$ and $S[V]$, the semijoin $R^T \bowtie S^T$ is defined as $\pi_U(R^T \bowtie S^T)$.

Join trees can now be processed by computing semijoins bottom-up

$\leadsto$ Yannakakis' Algorithm
Yannakakis' Algorithm by Example

Polynomial time procedure for answering BCQs

Does not immediately compute answers in the version given here

~ modifications needed

Even tree queries can have exponentially many results, but each can be computed (not just checked) in \( P \)

~ output-polynomial computation of results

Summary and Outlook

Conjunctive queries (CQs) are an important special case of FO queries

Boolean CQ answering, the homomorphism problem and constraint satisfaction problems are equivalent and \( NP \)-complete

CQ answering is simpler, namely in \( P \), when CQs are tree queries

- Check acyclicity with GYO algorithm
- Evaluate query using Yannakakis’ Algorithm

Open questions:

- Tree queries are rather special. Are there more general conditions for good queries?
- What about query optimisation?