Outline

1. Conjunctive Query Answering in DLs
2. Query Answering in Lightweight DLs
3. Query Answering in Expressive DLs
4. Summary

So far:

- Ontologies describe relevant terms and their relations
  - An inflammation of the lungs is a pneumonia
  - Streptococcus is a type of Gram positive Bacteria
  - Hypertension is a synonym for high blood pressure

- DL ABoxes store data
  - More important: actual data sources can be viewed as/mapped to ABoxes
  - patient(4971.120462) hasFinding(4971.120462, f14)
  - inflammation(f14) hasLocation(f14, lung)
  - hasCausitiveAgent(f14, strepPn) strepBacteria(strepPn)
Queries in DLs

- The user can pose queries over the vocabulary of the ontology, the system performs reasoning to return all answers

- Retrieve antibiotics that can be used to treat Gram-positive bacterial pneumonia
- Determine whether patient 6771.120884 has a close relative that is allergic to penicillin
- Retrieve all patients diagnosed with bacterial pneumonia that have an antibiotic allergy, or have a direct relative that has an antibiotic allergy

Queries in DLs (cont’d)

For example,

The (similar) query

Determine whether there exists a common allergy for some relative of patient 6771.120884

amounts to the FOL formula

\[
\text{Patient}(6771.120884) \land \exists x, y. (\text{hasRelative}(6771.120884, x) \land \text{hasAllergy}(x, y) \land \text{hasAllergy}(6771.120884, y))
\]

which is not equivalent to any DL concept

- DL expressions are poor query languages!

Query Languages (cont’d)

We want to access and process data using database inspired query languages that allow to flexibly select and join pieces of information

- Such queries are, in general, not expressible in DLs
- The existing algorithms and complexity results do not apply for them

We need new reasoners, new reasoning techniques, new algorithms, and new complexity bounds

We briefly discuss some of them in the rest of this lecture
DBs and Complete information

In traditional databases it is assumed that information is **complete**

For example, if the train departure table contains only

<table>
<thead>
<tr>
<th>Destination</th>
<th>Departure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Innsbruck</td>
<td>08:10</td>
</tr>
<tr>
<td>Innsbruck</td>
<td>10:10</td>
</tr>
<tr>
<td>Innsbruck</td>
<td>12:10</td>
</tr>
<tr>
<td>Innsbruck</td>
<td>14:10</td>
</tr>
<tr>
<td>Innsbruck</td>
<td>17:10</td>
</tr>
</tbody>
</table>

then we know that there is no train to Innsbruck departing at 9:05

KBs and Incomplete information

In contrast, in a DL KB we do not assume that information is complete

For example, if we only have

\[ \text{Person}(Andrea) \land \text{Female}(Maria) \]

then it does not imply that Maria is not a person, not that Andrea is neither male nor female.

In fact, incomplete information results in different models that have to be taken into account when answering queries

If in the example above we also have

\[ \text{person} \sqsubseteq \text{male} \sqcup \text{female} \]

then we will have at least two models: one where Andrea is male, and one where Andrea is female.
Open vs. Closed World Assumption

Formally, we have

- In Databases we make the closed word assumption (CWA): the facts that are not known to be true are considered false
- In contrast in DLs, like in standard first order logic, we make the open word assumption (OWA): a fact whose truth we know nothing about can be either true or false

This semantic difference has a huge impact on query answering!

Query answering in DLs vs. query answering in DBs

- A DB is a relational structure
- A KB represents a set of relational structure, its models

Given a KB $\mathcal{K}$ and a query $q$, we are interested in the certain answers to $q$ over $\mathcal{K}$, i.e., in the answers that occur in every model of $\mathcal{K}$.

Closely related to query answering in incomplete databases

- In Databases, the query is evaluated over one structure
  ~ model checking is computationally easy
- In DLs and incomplete DBs, the query is answered over many structures
  ~ logical consequence is computationally costly

Querying Knowledge Bases

The most important reasoning problem is query answering

Query Answering

Given a KB $\mathcal{K}$ and a query $q$, compute the tuples of individuals that are an answer for $q$ in every model of $\mathcal{K}$.

- We will define the notion of answer formally once we have formally defined the query language
- We sometimes consider queries with no answer variables, for which the answer is true if the query is true in all models, or false otherwise

Querying Knowledge Bases - Example

$$
\text{TBox } T: \begin{cases}
\exists \text{hasFather} . T & \text{Person} \\
\neg \exists \text{hasFather} . T & \text{Person} \\
\text{Person} & \neg \exists \text{hasFather}
\end{cases}
$$

$$
\text{ABox } A: \begin{cases}
\text{Person(john), Person(nick), Person(toni)} \\
\text{hasFather(john,nick), hasFather(nick,toni)}
\end{cases}
$$

$$
\begin{align*}
q_1(x,y) & \leftarrow \text{hasFather}(x,y) \\
q_2(x) & \leftarrow \exists y . \text{hasFather}(x,y) \\
q_3(x) & \leftarrow \exists y_1, y_2, y_3 . \text{hasFather}(x,y_1) \land \text{hasFather}(y_1,y_2) \\
& \land \text{hasFather}(y_2,y_3) \\
q_4(x,y_3) & \leftarrow \exists y_1, y_2 . \text{hasFather}(x,y_1) \land \text{hasFather}(y_1,y_2) \\
& \land \text{hasFather}(y_2,y_3)
\end{align*}
$$

Certain answers:

$$
\begin{align*}
cert(q_1, (T, A)) & = \{ \text{john,nick), (nick,toni)} \} \\
cert(q_2, (T, A)) & = \{ \text{john, nick, toni} \} \\
cert(q_3, (T, A)) & = \{ \text{john, nick, toni} \} \\
cert(q_4, (T, A)) & = \{ \}
\end{align*}
$$
Choosing a query language

We want to access and process data using database inspired query languages that allow to flexibly select and join pieces of information

Which is the best query language?

Some candidates:
- DL expressions: concepts and roles
  - They allow us to do simple instance queries
  - but as we have discussed, have very limited expressive power
- Formulas in FOL
  - A natural candidate - recall examples above
  - but they are not decidable
  - answering yes/no queries over an empty KB amounts to deciding FOL validity

Choosing a query language (cont’d)

A good alternative:
- Conjunctive Queries (CQs)
  - A special kind of positive existential FOL formulas
  - Equivalent to the plain Select-Project-Join fragment of SQL
  - Very popular in databases, standard language in many areas
  - Around 90% of the queries in actual applications fall in this fragment
  - Positive computational features
  - All the mentioned examples are CQs

They have been extensively studied for a wide range of DLs

An Example Conjunctive Query

\[ q(x) \leftarrow \text{Hero}(x), \text{hasMother}(x, v_1), \text{hasAncestor}(v_1, v_2), \text{Deity}(v_2) \]

Or, using standard FOL syntax:

\[ q(x) \leftarrow \exists v_1, v_2. \text{Hero}(x) \land \text{hasMother}(x, v_1) \land \text{hasAncestor}(v_1, v_2) \land \text{Deity}(v_2) \]

The query asks for the heroes \( x \) that have a divine ancestor on the maternal side

(we use red to highlight the answer variables)

Other query languages that have been studied for some DLs

Unions of Conjunctive queries (UCQs): disjunctions of CQs

\[ q(x) \leftarrow \{ \text{Hero}(x), \text{hasMother}(x, v_1), \text{hasAncestor}(v_1, v_2), \text{Deity}(v_2) \} \]

\[ \cup \{ \text{hasWife}(x, v_1), \text{Deity}(v_1) \} \]

Or, in FOL syntax

\[ q(x) \leftarrow (\exists v_1, v_2. \text{Hero}(x) \land \text{hasMother}(x, v_1) \land \text{hasAncestor}(v_1, v_2) \land \text{Deity}(v_2)) \lor (\exists v_1. \text{hasWife}(x, v_1) \land \text{Deity}(v_1)) \]

heroes that have a divine ancestor on the maternal side or are married with a goddess

They are also very popular, and they preserve many of the good computational properties of CQs
Other query languages (cont’d)

- Positive queries: positive FOL formulas

\[ q(x_1, x_2) \leftarrow \exists v. \text{hasRelative}(x_1, x_2) \land \text{hasChild}(x_1, v) \land \text{hasChild}(x_2, v) \land \text{Male}(x_1) \land \text{Female}(x_2) \land (\text{Mortal}(x_1) \lor \text{Mortal}(x_2)) \]

pairs of individuals who are relatives, have a common child \(v\), and at least one of them is mortal

They have the same expressiveness as UCQs, but they are more succinct

Other query languages (cont’d)

- Queries that allow to express more complex navigations on the database, in the style of XPath

\[ q(x_1, x_2) \leftarrow \text{hasParent}^* \circ \text{hasParent}^{-*}(x_1, x_2) \]

pairs of individuals who are relatives

- Combinations of the above

\[ q_1(x_1, x_2) \leftarrow \exists v. \text{hasParent}^* \circ \text{hasParent}^{-*}(x_1, x_2) \land \text{hasChild}(x_1, v) \land \text{hasChild}(x_2, v) \land \text{Male}(x_1) \land \text{Female}(x_2) \land (\text{Mortal}(x_1) \lor \text{Mortal}(x_2)) \]

pairs of individuals who are relatives, have a common child \(v\), and at least one of them is mortal

Outline

1. Conjunctive Query Answering in DLs
2. Query Answering in Lightweight DLs
3. Query Answering in Expressive DLs
4. Summary

Conjunctive Queries

A **conjunctive query** is a formula of the form

\[ q(\vec{t}) = \exists \vec{v}. A_1(\vec{v}_1) \land \ldots \land A_n(\vec{v}_n) \]

where

- \(\vec{t}\) and \(\vec{v}\) are lists of constants and variables,
- the \(A_i\) are concepts/roles,
- the \(\vec{v}_i\) are lists of arguments of matching arity,
- and \(\vec{v}_i \subseteq \vec{t} \cup \vec{v}\) for each \(i\).

We often write conjunctive queries as lists (or even sets) of atoms

\[ q(\vec{t}) = A_1(\vec{v}_1), \ldots, A_n(\vec{v}_n) \]

To define query answers, we use the notion of **match**
Query Match - Example

Intuitively, a match for a query \( q \) is an assignment of query variables to elements in an interpretation that makes \( q \) true. The answer to \( q \) is given by the image of the answer variables.

\[
q_2(x) \leftarrow \text{Hero}(x), \text{hasMother}(x, v_1), \text{hasAncestor}(v_1, v_2), \text{Deity}(v_2)
\]

Answer: heracles

Query Answering

Query answering consists on listing the answers to a query, i.e., it is an enumeration problem:

Definition (Query answering problem)

Given a KB \( K \) and a query \( q \) over \( K \), list all the tuples \( \vec{c} \) of constants such that \( \vec{c} \in \text{cert}(q, K) \).

When studying the complexity of query answering, we need to consider the associated decision problem:

Definition (Recognition problem for query answering)

Given a KB \( K \), a query \( q \) over \( K \), and a tuple \( \vec{c} \) of constants, check whether \( \vec{c} \in \text{cert}(q, K) \).

Entailment of Boolean Queries

Definition (Boolean query)

A query \( q \) over \( K \) that has no answer variables is called Boolean.

For a Boolean query \( q \), the main reasoning task is deciding whether \( q \) evaluates to \( \text{true} \) in all models:

Definition (Query entailment problem)

For a Boolean query \( q \) and a KB \( K \), we write \( K \models q \) if there is a match for \( q \) in every model of \( K \).

Given a KB \( K \) and a Boolean query \( q \) over \( K \), the query entailment problem is to decide whether \( K \models q \).
Query Answering and Query Entailment

- The recognition problem for query answering reduces to the entailment problem for Boolean queries:
  - Simply instantiate the query with the input tuple and verify the entailment of the resulting Boolean query
- Many algorithms focus on query entailment only
- Query answering can then be achieved by calling the query entailment procedure for each possible tuple (only constants occurring in the KB, thus finitely many tuples)
- In practice, of course, listing query answers should be done with smarter algorithms

Query answering in lightweight DLs

CQ entailment has been studied for many DLs.

For lightweight DLs like the DL-Lite and $\mathcal{EL}$ families, focus on:

- data complexity, which is usually tractable
- practical techniques for query answering with large amounts of data
  - query answering using existing technologies
  - in particular, using reductions into SQL and existing RDBMSs
  - or using other existing database technologies, such as Datalog engines

Recently, this kind of techniques have been explored for more expressive Horn DLs.

Query answering in expressive DLs

For expressive DLs that extend $\mathcal{ALC}$

- data complexity is typically coNP-complete
- the landscape for combined complexity of query answering is not so simple
- worst-case optimal algorithms are hard to come about
- until now, many decidability/complexity results obtained, but no practical algorithms implemented

Complexity of CQ entailment in DLs

<table>
<thead>
<tr>
<th></th>
<th>Combined complexity</th>
<th>Data complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain databases</td>
<td>NP-complete</td>
<td>in $\text{AC}_d$</td>
</tr>
<tr>
<td>DL-Lite</td>
<td>NP-complete</td>
<td>in $\text{AC}_d$</td>
</tr>
<tr>
<td>$\mathcal{EL}$</td>
<td>NP-complete</td>
<td>$\text{P}$-complete</td>
</tr>
<tr>
<td>Horn-$\mathcal{SHIQ}$</td>
<td>$\text{ExpTime}$-complete</td>
<td>$\text{P}$-complete</td>
</tr>
<tr>
<td>$\mathcal{SHIQ}$</td>
<td>2$\text{ExpTime}$-complete$^{(1)}$</td>
<td>coNP-complete$^{(2)}$</td>
</tr>
<tr>
<td>$\mathcal{SHOIQ}$</td>
<td></td>
<td>decidability open</td>
</tr>
</tbody>
</table>

(1) CQ answering is already 2ExpTime-hard for $\mathcal{ALCI}$ and $\mathcal{SH}$. $\mathcal{SHOIQ}$ and $\mathcal{SHOQ}$ are also in 2ExpTime.

(2) Already for TBoxes with a single disjunction in fragments of $\mathcal{ALC}$
### Query Answering vs. standard Reasoning

#### Instance checking

<table>
<thead>
<tr>
<th></th>
<th>Combined complexity</th>
<th>Data complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL-Lite</td>
<td>in P</td>
<td>in AC$_0$</td>
</tr>
<tr>
<td>(\mathcal{EL})</td>
<td>P-complete</td>
<td>P-complete</td>
</tr>
<tr>
<td>horn-(\mathcal{SHIQ})</td>
<td>ExpTime-complete</td>
<td>P-complete</td>
</tr>
<tr>
<td>(\mathcal{SHIQ})</td>
<td>ExpTime-complete</td>
<td>coNP-complete</td>
</tr>
<tr>
<td>(\mathcal{SHOIQ})</td>
<td>NExpTime-complete</td>
<td>coNP-hard</td>
</tr>
</tbody>
</table>

#### Query answering

<table>
<thead>
<tr>
<th></th>
<th>Combined complexity</th>
<th>Data complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL-Lite</td>
<td>NP-complete</td>
<td>in AC$_0$</td>
</tr>
<tr>
<td>(\mathcal{EL})</td>
<td>NP-complete</td>
<td>P-complete</td>
</tr>
<tr>
<td>horn-(\mathcal{SHIQ})</td>
<td>ExpTime-complete</td>
<td>P-complete</td>
</tr>
<tr>
<td>(\mathcal{SHIQ})</td>
<td>2ExpTime-complete</td>
<td>coNP-complete</td>
</tr>
<tr>
<td>(\mathcal{SHOIQ})</td>
<td>dec. open</td>
<td>coNP-hard</td>
</tr>
</tbody>
</table>

### Query Answering with Relational Database Systems

- Existing Relational Database Systems seem the most promising approach for achieving **scalability** of query answering.

- Main challenge to be overcome:
  - How do we make a RDBMS aware of the TBox?
  - Option 1: incorporate the TBox into the query → **query rewriting**
  - Option 2: incorporate the TBox into the ABox → **data completion**

These two approaches are analogous to **backward chaining** and **forward chaining** in automated deduction and logic programming.

### The Query Rewriting Approach

Given a KB \(\langle T, A \rangle\) and a CQ \(q\), obtain a FOL query \(\text{perfRef}_T(q)\) such that for every tuple \(\vec{a}\) of constants,

\[
\vec{a} \text{ is an answer for } q \text{ over } \langle T, A \rangle \iff \text{perfRef}_T(q) \text{ over } A
\]

\(\vec{a}\) is an answer for \(\text{perfRef}_T(q)\) over \(A\) (using the usual DB semantics)

- The ABox is stored directly as a database
- The query \(\text{perfRef}_T(q)\) is then evaluated over this DB using an off-the-shelf RDBMS (FOL queries are equivalent to SQL)
- This approach was introduced by Calvanese et.al. for DL-Lite
The Query Rewriting Approach – Example 1

TBox $\mathcal{T}$: $B' \sqsubseteq B$
$\exists S.T \sqsubseteq A$

Query: $q \leftarrow A(x), R(x, y), B(y)$

The rewriting of $q$ is the disjunction of:

- $A(x), R(x, y), B(y)$;
- $A(x), R(x, y), B'(y)$;
- $S(x, z), R(x, y), B(y)$;
- $S(x, z), R(x, y), B'(y)$;

- For DL-Lite, the perfect reformulation $\text{perfRef}_T(q)$ of a CQ $q$ (or a UCQ) is a UCQ
- Intuitively, we exploit the GCIs to obtain new queries that can contribute to the answer

Unbound variables and query reduction

Suppose our TBox contains $\text{Professor} \sqsubseteq \exists \text{Teaches}$.
The query $q_1(x) \leftarrow \text{Teaches}(x, y)$ can be rewritten as $q'_1(x) \leftarrow \text{Professor}(x)$

But this GCI can not be used for rewriting the queries $q_2(x, y) \leftarrow \text{Teaches}(x, y)$, $q_3 \leftarrow \text{Teaches}(\text{Peter}, y), \text{Teaches}(\text{John}, y)$

What about the query $q_4(x) \leftarrow \text{Teaches}(x, y), \text{Teaches}(z, y)$?

We should rewrite using the GCI, but for this we need to unify $x$ and $z$!

Unbound variables and query reduction (2)

A variable is bound if it is an answer variable or it it occurs in more than one atom. Otherwise, it is unbound.

The rules for existentials can only be applied to atoms with unbound variables

Unification may make unbound a variable that was bound

For the query rewriting algorithm to be complete, we need to exhaustively unify variables whenever possible

This unification step, often called reduction, is one of the most expensive components of the algorithm!
Query Rewriting and Data Complexity

- Answering FOL queries in standard DBs is in AC\(_0\) w.r.t. data complexity
- In query rewriting
  - The data, which is the only measured input, is not changed
  - The size of the rewritten query \(\text{perfRef}_T(q)\) depends only on \(q\) and \(T\), hence its fixed in data complexity
- Hence, the overall data complexity of answering a given \(q\) w.r.t. to \(\langle T, A \rangle\) is the same as evaluating a UCQ (or a FOL query) over \(A\).
- That is, it gives an AC\(_0\) upper bound

Combined complexity of the Perfect Reformulation Algorithm

Consider the TBox

\[ A \sqsubseteq B \]

and the query \(q \leftarrow B(x_1), B(x_2), \ldots, B(x_n)\)

The perfect reformulation algorithm gives \(2^n\) queries!

This exponential blow-up can be avoided, e.g.,

- Rewriting into a FOL query
  \[ q \leftarrow (A(x_1) \lor B(x_1)) \land (A(x_2) \lor B(x_2)) \land \cdots \land (A(x_n) \lor B(x_n)) \]
- Or rewriting into a non-recursive Datalog program
  \[ q \leftarrow B(x_1), B(x_2), \ldots, B(x_n) \]
  \[ B(x) \leftarrow A(x) \]

How well does Perfect Reformulation work in practice?

Thanks to DL-Lite and the PerfectRef algorithm, the actual realization of OBDA (Ontology Based Data Access) seemed possible

- First OBDA systems where developed building on this algorithm (e.g., QuOnto)
- However, it soon became apparent that they did not scale up as expected
- In real life ontologies, \(\text{perfRef}_T(q)\) may be so large that standard DBMS can not handle it!
  - The original QuOnto could only handle queries with up to 7-10 atoms

Improving over the Perfect Reformulation algorithm

Many optimizations and improved algorithms for DL-Lite have been proposed since then:

- More recent algorithms do not rewrite into a UCQ, but into compact FOL/Datalog queries
- Presto improves the original PerfectRef by rewriting into non-recursive Datalog, and avoiding unnecessary reduction steps
- Other algorithms for related languages (Datalog\(^+\), Horn-SHIQ, etc.) also rewrite into Datalog
- Techniques like semantic indexing compactly store TBox information in the data
Complexity of query rewriting

On the more theoretical side, much research efforts devoted to understand when small rewritings are possible:

- For plain DL-Lite, polytime procedure for query rewriting (Kikot et al., DL’11)
- For DL-Lite\(_R\), that allows role inclusions \( R \sqsubseteq S \) and underlies OWL 2 QL:
  - no polytime procedure for FO query rewriting (unless P=NP) (Kikot et al., DL’11)
  - polynomial non-recursive Datalog rewritings possible (under some assumptions), but resulting program complex (Gottlob & Schwentick, DL’11, KR’12)
  - Detailed analysis of when polynomial FO rewritings are possible (Kikot et al., KR’12)

The limits of Query Rewriting

Recall that the pure query rewriting approach gives an \( \text{AC}_0 \) upper bound for data complexity

Hence, the rewriting approach (into FOL queries) can only work for DLs whose data complexity is in \( \text{AC}_0 \)

That is, we can only use it for the DL-Lite family

\( \mathcal{EL} \) and Query Rewriting

TBox \( T: \exists S.A \sqsubseteq A \)

Query: \( q \leftarrow A(x) \)

The rewriting of \( q \) is the disjunction of:

\[
A(x) \\
S(x, y_1), A(y_1) \\
S(x, y_1), S(y_1, y_2), A(y_2) \\
S(x, y_1), S(y_1, y_2), S(y_2, y_3), A(y_3) \\
\ldots
\]

This can not be written as a finite SQL query!

It can be written as \( S^*(x, y), A(y) \), but reflexive-transitive closure is not FOL-expressible!

\( \mathcal{EL} \) is P-hard in data complexity, hence we can not use the Query Rewriting approach as defined above

Query Rewriting beyond DL Lite

First-order rewritability fails for every DL beyond DL Lite

Some possible solutions:

1. Rewrite into query language that is more expressive than FOL
   - Every CQ over \( \mathcal{EL} \) can be rewritten into a Datalog query

   \[
   q(x) : -A(x) \\
   A(x) : -S(x, y), A(y)
   \]

   - Rewriting into Datalog works even for Horn-SHITQ, the most expressive DL for which query answering has been implemented

2. Give up the data independence of the rewriting approach, and modify also the ABox (Lutz et al. 08)
The Data completion approach – Naive attempt

Basic idea: add the TBox information to the ABox

- For each lightweight DL KB there is one canonical model that can be used for answering all queries
- If we represent that canonical model as a database, then we can simply pose queries to it
- But often this does not work:
  - the canonical model for query answering may be infinite, even for DL Lite and $\mathcal{EL}$!

The combined approach

One solution is not to realize the full canonical model, but instead to combine data completion with query rewriting

The combined approach (Lutz et al. 09)

Idea: Given a KB $\langle T, A \rangle$ and a CQ $q$, obtain a FOL query $q'$ and a polynomial ABox $A'$ such that for every tuple $\vec{a}$ of constants,

$\vec{a}$ is an answer for $q$ over $\langle T, A \rangle$ iff $\vec{a}$ is an answer for $q'$ over $A'$ (using the usual DB semantics)

This requires the data to be modified
- Assumes a different setting (e.g., access to the data a priory, privacy)

The Combined approach – An overview

- Instead of the real canonical model, we realize in the database another representative model that reuses existentially quantified elements
- Reusing elements may introduce spurious query matches
  - that is why we need to rewrite the query as well
- With suitable rewritings, we can obtain a query that has a match in the small model iff it has a match in the canonical one

How good is the Combined approach in practice?

- This approach has been successfully applied to $\mathcal{EL}$ and DL-Lite
- Although the completed data is polynomial in the input, it may be very large in practice
- In fact, the data completion stage is very expensive and may imply several hours of offline processing
- In many cases, it may not even be applicable: e.g., if one has no access to the data a priori, or no right to modify it
Query Answering in Horn-\textit{SHIQ}

- The tractability of query answering is considered one of the most useful features of DL-Lite and $\mathcal{EL}$
- This positive feature is preserved in the Horn fragments of more expressive DLs like \textit{SHIQ}
  - Horn-\textit{SHIQ} is \textit{tractable in data complexity} (PTime-complete)
  - The combined complexity is not higher than for standard reasoning
    - ExpTime-complete

\section*{coNP-hardness in Data Complexity}

This is lost as soon as we allow one single disjunction in the TBox.

\textbf{2+2 UNSAT}

Instance: A formula in CNF

\[ \varphi = c_1 \land \ldots \land c_n \]

over variables $x_1, \ldots, x_m$ and constants \texttt{true}, \texttt{false}, where each $c_i$ is of the form $v_{i,1} \lor v_{i,2} \land \lnot v_{i,3} \land \lnot v_{i,4}$.

Question: Is $\varphi$ unsatisfiable?

We show coNP-hardness this by a reduction from the 2+2 UNSAT problem.

\section*{Canonical models}

Intuitively, the main reason why tractability is preserved in the absence of disjunction is that there exists a \textit{canonical model}
- minimal model which gives correct answers to all queries
- consists of two parts:
  - \textit{core part} (ABox individuals)
  - \textit{anonymous part} (possibly infinite trees attached to the core)

Like for DL-Lite and $\mathcal{EL}$, we can also rely on this canonical model for Horn fragments of expressive DLs
Query Answering for Horn-\textit{SHIQ}

We now discuss a query rewriting algorithm for Horn-\textit{SHIQ}.

- We rewrite \( q \) into a UCQ \( \text{Rew}_T(q) \) (depends on the TBox \( T \)).
- \( T \) is rewritten into a set of Datalog rules \( \text{Comp}(T) \).
- Answering \( q \) over \((T, A)\) amounts to evaluating the Datalog program \( A \cup \text{Comp}(T) \cup \text{Rew}_T(q) \).

We can also evaluate \( \text{Rew}_T(q) \) over the core part of the canonical model (with no anonymous part).

\( \text{Rew}_T(q) \) can be exponential, but has manageable size for real queries and ontologies.

The rewriting algorithm

Main idea:

- Eliminate query variables that can be matched at unnamed objects.
  - Query matches have tree-shaped parts.
  - We clip off the variables \( x \) that can be leaves.
  - Replace them by constraints \( D(y) \) on their parent variables \( y \).
  - The added atoms \( D(y) \) ensure the existence of a match for \( x \).

- In the resulting queries all variables are matched to named objects.

One step of query rewriting

The query using the axiom is rewritten to

Another step of query rewriting

To handle transitive roles in the query:

- introduce a new variable between eliminated variable and some of its predecessors.
- eliminate sets of variables (variables connected in the query may be mapped to same object).
The Query Answering Algorithm

Algorithm 1: Answering CQs via Query Rewriting

\[\text{Input: normal Horn-}SHIQ\ \text{KB } K = (T,A), \text{ Conjunctive Query } q\]

\[\text{Output: query answers}\]

\[Sat(T) \leftarrow \text{Saturate}(T);\]
\[Rew_T(q) \leftarrow \text{Rewrite}(q, Sat(T));\]
\[Comp(T) \leftarrow \text{CompletionRules}(T);\]
\[P \leftarrow A \cup \text{Comp}(T) \cup Rew_T(q);\]
\[\text{ans} \leftarrow \{\vec{u} \mid q(\vec{u}) \in \text{MinimalModel}(P)\}; \quad \triangleright \text{ call Datalog reasoner}\]

- We use a calculus to compute the saturated set of all axioms implied by \(T\)
- The completion rules \(\text{Comp}(T)\) are straightforward, e.g.
  \[B(y) \leftarrow A(x), r(x,y) \quad \text{for each } A \sqsubseteq \forall r.B \in T\]
  \[r(x,y) \leftarrow r_1(x,y), \ldots, r_n(x,y) \quad \text{for each } r_1 \sqcap \ldots \sqcap r_n \sqsubseteq r \in T\]

Recalling the Complexity of CQ entailment

<table>
<thead>
<tr>
<th></th>
<th>Combined complexity</th>
<th>Data complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain databases</td>
<td>NP-complete</td>
<td>in (AC_0)</td>
</tr>
<tr>
<td>DL-Lite</td>
<td>NP-complete</td>
<td>in (AC_0)</td>
</tr>
<tr>
<td>(\mathcal{EL})</td>
<td>NP-complete</td>
<td>P-complete</td>
</tr>
<tr>
<td>Horn-(SHIQ)</td>
<td>ExpTime-complete</td>
<td>P-complete</td>
</tr>
<tr>
<td>(SHIQ)</td>
<td>2ExpTime-complete</td>
<td>coNP-complete (^{(2)})</td>
</tr>
<tr>
<td>(SHOIQ)</td>
<td>decidable</td>
<td>open</td>
</tr>
</tbody>
</table>

Querying Graph Databases

- The problem of querying graph databases has been extensively studied in the database community.
- It is similar to query answering in DLs (graph databases are just ABoxes), but without a TBox

The problem we study in DLs is more general, but our results have a major weakness:

- Most DL research concerns simple conjunctive queries (CQs):
  - \(\Rightarrow\) too poor for graph databases and data on the web!
- The fundamental language is regular path queries (RPQs).

Expressive Regular Path Queries

- In RPQs, binary atoms allow us to search for arbitrarily long paths between two individuals that comply to a regular expression.
- Called \textit{conjunctive}, if we can have several atoms.
- Called \textit{2-way} if we can use both role names and their inverses.
- Sometimes, the test operator is used to search for explicit concept labels along the path

This kind of path navigation is present in:

- XPath, the navigational core of the XQuery language for XML
- New versions of SPARQL, the query language for RDF:
  - extensions like nSPARQL and vSPARQL
  - path properties in SPARQL 1.1 (possibly the next standard)
Enriching graph databases with DL Ontologies

We have discussed that a DL TBox, we can enrich the data with ontological constraints:

- give a conceptual model of the application domain
- provide an enriched vocabulary for querying

(1) \( \exists \text{worksOn} . \text{CompSci} \sqsubseteq \text{CompScientist} \)
(2) \( \exists \text{wroteThesis}.(\exists \text{hasTopic}.\text{CompSci}) \sqsubseteq \text{CompScientist} \)
(3) \( \exists \text{worksOn}. \text{MathLogic&Foundns} \sqsubseteq \text{Logician} \)
(4) \( \exists \text{wroteThesis}.(\exists \text{hasTopic}.\text{MathLogic&Foundns}) \sqsubseteq \text{Logician} \)
(5) \( \text{MathLogic&Foundns} \sqsubseteq \text{General&Foundns} \)
(6) \( \text{CompSci} \sqsubseteq \text{AppliedMath&Other} \)
(7) \( \text{Physics} \sqsubseteq \text{AppliedMath&Other} \)
(8) \( \text{General&Foundns} \sqsubseteq \text{Subject} \)
(9) \( \text{DiscreteMath&Algebra} \sqsubseteq \text{Subject} \)
(10) \( \text{AppliedMath&Other} \sqsubseteq \text{Subject} \)

Examples of Regular Path Queries

The following atom navigates a chain of advisors that are computer scientists or logicians, until a biologist is reached.

\[
(\text{hasAdvisor} \circ \text{Logician}? \cup \text{hasAdvisor} \circ \text{CompScientist})^* \\
\circ \text{hasAdvisor} \circ \text{Biologist}? (x, y)
\]

Using a conjunction of atoms, we can find scientists \( x \) that have a student \( y \) and a co-author \( z \) sharing such an advisor \( u \):

\[
\text{hasAdvisor}^{-} (x, y), \; \text{hasCoauthor}(x, z), \\
(\text{hasAdvisor} \circ \text{Logician}? \cup \text{hasAdvisor} \circ \text{CompScientist})^* \\
\circ \text{hasAdvisor} \circ \text{Biologist}? (y, u), \\
(\text{hasAdvisor} \circ \text{Logician}? \cup \text{hasAdvisor} \circ \text{CompScientist})^* \\
\circ \text{hasAdvisor} \circ \text{Biologist}? (z, u)
\]

Regular Path Queries in DLs

Conjunctive regular path queries seem better than (U)CQs as a standard query language in DLs
but maybe it is too expensive to support regular paths?

Good news:

- Decidable even for very expressive DLs like SRIQ, ZIQ and Horn-SROIQ
- From Horn-SRIQ to full ZIQ, ZOQ, ... the complexity is not higher than for plain CQs!

...but for these expressive languages, query answering is very hard and no implementations so far.
Expressive Regular Path Queries in Lightweight DLs

- RPQs and their extensions have only been explored very recently for the lightweight DLs that are popular for OBDA, like DL-Lite and EL.

- For C(2)RPQs, we now have the following complexity results:

<table>
<thead>
<tr>
<th></th>
<th>Data complexity</th>
<th>Combined complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL-LiteRDFS</td>
<td>NLSpace-complete</td>
<td>NP-complete</td>
</tr>
<tr>
<td>DL-LiteR</td>
<td>NLSpace-complete</td>
<td>PSpace-complete</td>
</tr>
<tr>
<td>EL,ELH</td>
<td>P-complete</td>
<td>PSpace-complete</td>
</tr>
</tbody>
</table>

harder than plain CQs [AC0]  
harder than RPQs over graph DBs [NP]

Lower Bounds

<table>
<thead>
<tr>
<th></th>
<th>Data complexity</th>
<th>Combined complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL-LiteRDFS</td>
<td>NLSpace-complete</td>
<td>NP-complete</td>
</tr>
<tr>
<td>DL-LiteR</td>
<td>NLSpace-complete</td>
<td>PSpace-complete</td>
</tr>
<tr>
<td>EL,ELH</td>
<td>P-complete</td>
<td>PSpace-complete</td>
</tr>
</tbody>
</table>

All hardness results hold for CRPQs (1-way)

1. known for plain RPQs over plain graph DBs: simple reduction from directed graph reachability using a query $r^+(x,y)$

2. well-known, already for standard reasoning

3. well-known for CQ entailment over plain DBs

4. reduction from emptiness of the intersection of regular languages $L_1,\ldots,L_n$ over alphabet $\Sigma$

PSpace-hardness of CRPQs

- Given an alphabet alphabet $\Sigma$, we can easily generate all words in $\Sigma^*$ using a TBox
  - We use each $r \in \Sigma$ as a role name, and an auxiliary concept name $A$
  - For DL-Lite, we use the following TBox:
    $$\mathcal{T}_\Sigma = \{ A \sqsubseteq \exists r \mid r \in \Sigma \} \cup \{ \exists r \sqsubseteq \exists s \mid r, s \in \Sigma \}$$
  - For EL, we can use an even simpler TBox:
    $$\mathcal{T}_\Sigma = \{ A \sqsubseteq \exists r.A \mid r \in \Sigma \}$$

- Given regular languages $L_1,\ldots,L_n$ over alphabet $\Sigma$, we define the query
  $$q = \exists x \ L_1(a,x) \land \ldots \land L_n(a,x)$$

  $$(\mathcal{T}_\Sigma, \{ A(a) \}) \models q \iff L_1 \cap \cdots \cap L_n \neq \emptyset$$

Membership Results

- The membership results are non-trivial

- They make extensive use of the existence of the canonical model

- Obtained via a complex adaptation of the Horn-SHIQ algorithm above
3. Query Answering in Expressive DLs

Outline

1. Conjunctive Query Answering in DLs
2. Query Answering in Lightweight DLs
3. Query Answering in Expressive DLs
4. Summary

Query Answering in Expressive DLs

Assume a given knowledge base $\mathcal{K}$ and a query $q$

- We want to decide $\mathcal{K} \models q$
- This is equivalent to deciding whether there is a countermodel witnessing $\mathcal{K} \not\models q$, i.e., a model of $\mathcal{K}$ where there is no match for $q$
- For most DLs, one can show that if $\mathcal{K} \not\models q$, then there is a countermodel that has some kind of forest-shaped

Query Matches in Forest-shaped models

A match for $q$ in a canonical model has two parts:

- a partial match into the A-Box part (roots)
- maps for subqueries inside the trees

Searching for Countermodels

- All existing algorithms search for a forest-shaped countermodel
- In many cases, this is done in three stages:
  1. Consider all partial matches of the query into the ABox part
  2. generate all combinations $Q$ of subqueries that contain some subquery generated by a partial match
  3. For each $Q$, decide existence of a tree-shaped model part $\mathcal{I}$ with $\mathcal{I} \not\models Q$
- The last step focuses on trees only, and is often achieved by elaborate adaptations of TBox reasoning techniques
What makes query answering hard?

For most expressive DLs, query answering is very hard:

- There are exponentially many possible ABox parts, and exponentially many partial matches in each of them.
- \( Q \) can be exponential in \( q \).
- A (subquery) can be matched to a tree in exponentially many different ways.
- Deciding \( I \models Q \) inside a tree is exponentially harder than standard reasoning.

Many algorithms for query answering in expressive DLs have been developed, but none of them seems implementable.

Complexity of Query Answering in Expressive DLs

<table>
<thead>
<tr>
<th>Logic</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{ALCQ} )</td>
<td>3ExpTime-complete</td>
</tr>
<tr>
<td>( \text{SHIQ} )</td>
<td>2ExpTime-complete</td>
</tr>
<tr>
<td>( \text{SROIQ} )</td>
<td>ExpTime-complete</td>
</tr>
<tr>
<td>( \text{ZOIQ} )</td>
<td>Undecidable</td>
</tr>
</tbody>
</table>

Overview of some Techniques for QA in Expressive DLs – Part 1

- **Modified tableaux** algorithms that take the query size into account when blocking:
  - Often called \( n \)-blocking or CARIN blocking
  - First introduced for \( \text{ALCN} \) (in the context of a language called CARIN) (Levy and Rousset 98)
  - Has been extended to more expressive logics, but does not work for DLs with transitive roles.

Overview of some Techniques for QA in Expressive DLs – Part 2

- **Tuple-graph** or rolling up techniques:
  - Each way of mapping a subquery inside a tree can be expressed as a DL concept
  - Using this, query answering can be reduced to satisfiability testing
    - exponentially many satisfiability tests
    - each of them receives an exponentially larger KB as input
  - First introduced for a DL called \( \text{DLR} \) (Calvanese et.al.98)
  - Yields optimal complexity bounds
  - Has been extended to other DLs like \( \text{ALCHQ} \), \( \text{SHIQ} \), and \( \text{SHOQ} \).
Overview of some Techniques for QA in Expressive DLs – Part 3

- Automata on infinite trees reduce the existence of a countermodel to the emptiness test of a suitable automaton
  - Yield optimal bounds for combined complexity
  - They can handle both the ABox and the tree part, but are not optimal in data complexity
  - Can be combined with other techniques
  - Can accommodate many constructs
  - They have been used to obtain complexity bounds for the most expressive decidable DLs so far

- Knot-based techniques focus on the 'tree part' of the problem
  - Use simple, local representations of models
  - Allow to obtain optimal complexity bounds some hard cases

Summary

Query answering is a relatively new DL reasoning task that is gaining importance in many applications

- In general, query answering in DLs is harder and more involved than:
  - Query Answering in plain DBs
  - Traditional reasoning in DLs

- For lightweight DLs:
  - the complexity seems manageable
  - most successful approaches rely on relational DBs
  - in practice, scalability not so easy
  - many open challenges

- For expressive DLs:
  - the problem is usually very hard
  - many questions are still open
  - no practical algorithms available