DLs and Data Access

Apart from describing terminologies (i.e., the structure of a domain), DLs allow to assert specific facts (data) about the domain.

For example,

- patient(4971.120462) hasFinding(4971.120462, f14)
- inflammation(f14) hasLocation(f14, lung)
- hasCausitiveAgent(f14, strepPn) strepBacteria(strepPn)

When DLs originated, data was not considered crucial, but in many applications it is data that matters most!

DL ontologies can be naturally seen as descriptions of data repositories.

Queries in DLs

DL-systems are now expected to support reasoning about queries.

The basic service of this kind is query answering.

- 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
- Estimate index of penicillin allergy among ethnic Asian male adults

Apart from data access, DL ontologies also find increasing attention as tools for data management, data integration, data exchange, etc.

E.g., in a medical domain, one may be interested in queries like:

- 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
- Estimate index of penicillin allergy among ethnic Asian male adults
Queries in DLs (cont’d)

- Other services may allow us to compare and process query outputs, or to compare and optimize queries

  - Estimate index of penicillin allergy among ethnic Asian male adults
  - Are all treatments effective for gram positive pneumonia also effective for all pneumococcal respiratory infections?
  - Does the (easy to evaluate) query Q1 always provide the same answer as the (harder to evaluate) query Q2?

Sometimes these reasoning services are reducible to instance checking

For example,

The query

\[ \text{Determine whether patient 6771.120884 has a close relative that is allergic to penicillin} \]

reduces to checking whether

\[ \mathcal{K} \models \text{Patient} \sqcap \exists \text{hasRelative.}(\exists \text{hasAllergy}.\text{Penicillin})(6771.120884) \]

But this holds only for the very simple queries that can be written as concepts

Queries in DLs (cont’d)

For example,

The (similar) query

\[ \text{Determine whether there exists a common allergy for some relative of patient 6771.120884} \]

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

For example, the following FOL expressions seem desirable, non-trivial queries to pose to a DL KB

\[
\begin{align*}
q_1 & \leftarrow \text{Patient}(6771.120884) \land \exists y, z. (\text{hasRelative}(6771.120884, y) \land \text{hasFinding}(y, z) \land \text{hasFinding}(6771.120884, z)) \\
q_2(y) & \leftarrow \text{Patient}(6771.120884) \land \exists z. (\text{hasRelative}(6771.120884, y) \land \text{hasFinding}(y, z) \land \text{hasFinding}(6771.120884, z)) \\
q_3(x) & \leftarrow \text{Patient}(x) \land \exists y, z. (\text{hasRelative}(x, y) \land \text{hasFinding}(x, z) \land \text{hasFinding}(y, z))
\end{align*}
\]

The variables in red the answer variables, aka distinguished or output variables
Query Languages (cont’d)

We want to access and processing 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.

In this lecture, we briefly discuss some of them.

Databases and DL Ontologies

- In a DL ontology, the ABox can be seen as a database.

- As for the TBox:
  - It is similar to a conceptual schema, but while the latter is only important in the design phase, the TBox will still be relevant when actually answering queries.
  - It expresses constraints on the schema, but the semantics is different from traditional DB constraints.

  - the constraints need not be satisfied by the ABox (database) at runtime!

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>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 (infinitely) 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
Query Answering in DLs

2. Query Answering in DLs

2.1 DL ontologies vs. databases

Querying Knowledge Bases - Example

**TBox** $\exists \text{hasFather}. \top \sqsubseteq \text{Person}$

**ABox** $\text{Person}(\text{john}), \text{Person}(\text{nick}), \text{Person}(\text{toni})$

$\text{hasFather}(\text{john}, \text{nick}), \text{hasFather}(\text{nick}, \text{toni})$

**Queries:**

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

**Certain answers:**

$\text{cert}(q_1, \langle T, A \rangle) = \{ (\text{john}, \text{nick}), (\text{nick}, \text{toni}) \}$
$\text{cert}(q_2, \langle T, A \rangle) = \{ \text{john}, \text{nick}, \text{toni} \}$
$\text{cert}(q_3, \langle T, A \rangle) = \{ \text{john}, \text{nick}, \text{toni} \}$
$\text{cert}(q_4, \langle T, A \rangle) = \{ \}$

Choosing a query language

We want to access and processing 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

Other query languages

Some other languages that are considered of interest are:

- **Unions of Conjunctive queries (UCQs): disjunctions of CQs**

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

heroes that have a divine ancestor on the maternal side or are married with a goddess
Other query languages (cont’d)

- Positive queries: positive FOL formulas

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

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

Conjunctive Queries

Conjunctive Query (CQ)

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

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

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

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

To define query answers, we use the notion of match.

**Query Match**

A match for \( q(t) \) in an interpretation \( I \) is a substitution from the variables and constants in \( q \) to \( \Delta^I \) such that:

- \( \pi(a) = a^I \) for each individual \( a \),
- \( \pi(x) \in A^I \) for each \( A(x) \in q \), and
- \( \langle \pi(x), \pi(y) \rangle \in r^I \) for each \( r(x, y) \in q \).

**Query Answer**

A tuple of individuals \( \langle a_1, \ldots, a_n \rangle \) is called an answer for \( q(t_1, \ldots, t_n) \) in an interpretation \( I \) if there is a match \( \pi \) for \( q \) in \( I \) such that \( \pi(t_i) = a_i^I \) for every \( i \).

The tuple \( \langle a_1, \ldots, a_n \rangle \) is called an answer for \( q(t_1, \ldots, t_n) \) over a KB \( K \) if it is an answer in every model \( I \) of \( K \).

**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 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 in 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 answering should be done with smarter algorithms

Complexity measures for query answering over KBs

When measuring the complexity of answering a query \( q(\vec{x}) \) over an KB \( \mathcal{K} = (\mathcal{T}, \mathcal{A}) \), various parameters are of importance.

Depending on which parameters we consider, we get different complexity measures:

- **Data complexity**: only the size of the ABox (i.e., the data) matters. TBox and query are considered fixed.
- **Query complexity**: only the size of the query matters. TBox and ABox are considered fixed.
- **Schema complexity**: only the size of the TBox (i.e., the schema) matters. ABox and query are considered fixed.
- **Combined complexity**: no parameter is considered fixed.

In the (common) setting where the size of the data largely dominates the size of the conceptual layer (and of the query), data complexity is the relevant complexity measure.

### Complexity of query answering in DLs

<table>
<thead>
<tr>
<th>DL</th>
<th>Combined complexity</th>
<th>Data complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain databases</td>
<td>NP-complete</td>
<td>in ( \mathcal{AC}_0 )</td>
</tr>
<tr>
<td>DL-Lite</td>
<td>NP-complete</td>
<td>in ( \mathcal{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>Expressive DLs</td>
<td>2ExpTime-complete(^{(1)})</td>
<td>coNP-complete(^{(2)})</td>
</tr>
</tbody>
</table>

\(^{(1)}\) For a few expressive DLs (e.g. \( \mathcal{ALC}HQ \)) CQ answering is still in ExpTime, but already 2ExpTime-hard for \( \mathcal{ALCI}, \mathcal{SH} \)
\( \mathcal{SHIQ}, \mathcal{SHOI} \) and \( \mathcal{SHOQ} \) are in ExpTime, for \( \mathcal{SHOIQ} \) and \( \mathcal{SROIQ} \) even decidability is open

\(^{(2)}\) Already for a TBox with a single disjunction, but still coNP-complete for very expressive DLs
Query Answering vs. standard Reasoning

<table>
<thead>
<tr>
<th>Instance checking</th>
<th>Combined complexity</th>
<th>Data complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL-Lite</td>
<td>in P</td>
<td>in AC₀</td>
</tr>
<tr>
<td>( \mathcal{E}L )</td>
<td>P-complete</td>
<td>P-complete</td>
</tr>
<tr>
<td>Horn-SHIQ</td>
<td>ExpTime-complete</td>
<td>P-complete</td>
</tr>
<tr>
<td>Expressive DLs</td>
<td>ExpTime-complete</td>
<td>coNP-complete</td>
</tr>
<tr>
<td>SHOIQ</td>
<td>NExpTime-complete</td>
<td>coNP-hard</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query answering</th>
<th>Combined complexity</th>
<th>Data complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL-Lite</td>
<td>NP-complete</td>
<td>in AC₀</td>
</tr>
<tr>
<td>( \mathcal{E}L )</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>Expressive DLs</td>
<td>2ExpTime-complete</td>
<td>coNP-complete</td>
</tr>
<tr>
<td>SHOIQ</td>
<td>dec. open</td>
<td>coNP-hard</td>
</tr>
</tbody>
</table>

Query answering in lightweight DLs

For lightweight DLs like the DL-Lite and \( \mathcal{E}L \) 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 a RDBMS and reductions into SQL over relational databases
  - also using other technologies, such as Datalog engines

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

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
The Query Rewriting Approach

This approach was introduced by Calvanese et al. for DL-Lite

**Idea:** Given a KB \( \langle T, A \rangle \) and a CQ \( q \), obtain a (SQL) query \( q_T \) 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 \( q_T \) over \( A \) (using the usual DB semantics)

This allows us to directly use off-the-shelf RDBMSs:
- The ABox is stored directly as a database
- The query \( q_T \) is then evaluated over this DB
- Optimal from the data complexity point of view, since it really optimizes query answering w.r.t. the data size

### The Query Rewriting Approach – Example 1

**TBox** \( T \):

\[
B' \sqsubseteq B
\]
\[
\exists S. T \sqsubseteq A
\]

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

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

\[
A(x), R(x, y), B(y)
\]
\[
A(x), R(x, y), B'(y)
\]
\[
S(x, z), R(x, y), B(y)
\]
\[
S(x, z), R(x, y), B'(y)
\]

- A CQ \( q \) is reformulated into a set of queries (UCQ) \( q_T \)
- Intuitively, we exploit the GCIs to obtain new queries that can contribute to the answer

### Query rewriting in DL-Lite

The rewriting algorithm is given as a set of rules that apply the GCIs in \( T \) (from left to right) to a given query:

\[
A_1 \sqsubseteq A_2 \quad \ldots \quad A_2(x) \ldots \leadsto \ldots \quad A_1(x)\ldots
\]
\[
\exists P \sqsubseteq A \quad \ldots \quad A(x) \ldots \leadsto \ldots \quad P(x, \_\_)
\]
\[
\exists P^{-} \sqsubseteq A \quad \ldots \quad A(x) \ldots \leadsto \ldots \quad P(\_, x)
\]
\[
A \sqsubseteq \exists P \quad \ldots \quad P(x, \_\_), \ldots \leadsto \ldots \quad \exists A(x)\ldots
\]
\[
A \sqsubseteq \exists P^{-} \quad \ldots \quad P(\_, x), \ldots \leadsto \ldots \quad \exists A(x)\ldots
\]
\[
\exists P_1 \sqsubseteq \exists P_2 \quad \ldots \quad P_2(x, \_\_), \ldots \leadsto \ldots \quad P_1(x, \_\_)
\]
\[
P_1 \sqsubseteq P_2 \quad \ldots \quad P_2(x, y), \ldots \leadsto \ldots \quad P_1(x, y)
\]

where \( \_ \) denotes an unbound variable, i.e., a fresh variable that appears only once

We obtain the rewritten \( q_T \) by applying the rules in every possible way.

### The limits of Query Rewriting

- Query answering 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 rewriting does not depend on the data
- Hence, the Query Rewriting approach only works for DLs whose data complexity is in \( AC_0 \)
- That is, we can only use it for the DL-Lite family
The Data completion approach – Naive attempt

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 this does not work:

\[ \sim \text{ the canonical model for query answering is infinite, even for DL Lite!} \]

The Data completion approach – An overview

Instead, we realize in the database another representative model that reuses existentially quantified elements.

For $\mathcal{EL}$, we can use the small canonical model that we discussed in the last lecture.

Reusing elements may introduce spurious query matches.

\[ \sim \text{ that is why we need to rewrite the query as well} \]

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.

This approach has been successfully applied to $\mathcal{EL}$ and DL-Lite.
Outline

1. DLs and Data Access
2. Query Answering in DLs
   2.1 Query answering in DL ontologies vs. query answering in databases
   2.2 Query Languages
3. Conjunctive Query Answering in DLs
   3.1 Reasoning about queries
   3.2 Lightweight DLs vs Expressive DLs
4. Query Answering in Lightweight DLs
   4.1 DL Lite and the Query Rewriting Approach
   4.2 Data Completion and $\mathcal{EL}$
5. QA in Expressive DLs
   5.1 What makes query answering hard?
   5.2 Techniques for QA in Expressive DLs
6. Summary

Overview of the Problem

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

We want to decide $\mathcal{K} \not\models q$

- Is there a countermodel for $q$ and $\mathcal{K}$?
  - i.e., a model of $\mathcal{K}$ where there is no match for $q$

- $\mathcal{K}$ usually has some kind of forest-shaped canonical models
- we only need to consider such models

Overview of the Problem (cont’d)

A match for $q$ in a canonical model may comprise:

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

We want to find a countermodel for $q$, i.e., a model of $\mathcal{K}$ where every partial match into the ABox generates some subquery that has no match in the corresponding tree

Strategy for finding a countermodel

Very roughly,

- For each way to complete the A-Box, we look at all the possible partial maps
- We consider all combinations $Q$ of subqueries that contain some subquery generated by each partial match
- For each combination $Q$, search for trees (for all roots) that avoid all the assigned subqueries, i.e., decide if there exists tree-shaped $I$ such that $I \not\models q$ for each $q \in Q$.

Summing up, algorithms (usually) have two key ingredients:

1. generate relevant combinations $Q$
2. decide existence of tree-shaped $I$ with $I \not\models Q$
What makes query answering hard?

Two general aspects to be handled when proving complexity bounds:

1. Size of the relevant combinations $Q$
2. Complexity of deciding the existence of a tree $I$ with $I \not\models Q$ (in what follows, $\text{Tree} \not\models Q$)

Rough intuition:
The complexity of $\text{Tree} \not\models q$ (for one $q \in Q$) depends on the number of different ways in which (subqueries of) $q$ may be mapped into a tree.

Usually 2 subsumes 1, and the worst-case complexity arises already for $\text{Tree} \not\models q$ (for one $q$).

---

Complexity of Query Answering in Expressive DLs

<table>
<thead>
<tr>
<th>DL</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{SHIQ}$</td>
<td>co-NExpTime-complete</td>
</tr>
<tr>
<td>$\text{ALCI}$</td>
<td>ExpTime-complete</td>
</tr>
<tr>
<td>$\text{ALCH}$</td>
<td>recursive</td>
</tr>
<tr>
<td>$\text{SRQ}$</td>
<td>2ExpTime-complete</td>
</tr>
<tr>
<td>$\text{SROQ}$</td>
<td>3ExpTime</td>
</tr>
<tr>
<td>$\text{SHIQ}$</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 Rouset 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: consider all possible ways of mapping the query into the forest shaped models of the KB, and build a concept for each of them. Then we test satisfiability of an exponentially larger KB that incorporates the negation of all concepts
  - First introduced for a DL called $\text{DLR}$ (Calvanese et.al. 98)
  - Yields optimal complexity bounds
  - Has been extended to other DLs, but the generalizations are not easy

---
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 combined complexity bounds
  - Extended rather naturally to more expressive logics and allow to obtain algorithms for very expressive DLs
  - Not good for obtaining optimal data complexity

- **Knot-based** and related techniques that use simple representations of models and work 'locally' on models
  - Allow to obtain optimal complexity bounds in the subtle cases
  - Can be conceptually rather simple
  - Optimal in data complexity

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

  *Anybody looking for nice research topics?*

- For lightweight DLs, on the other hand
  - scalable query answering is feasible
  - most successful approaches rely on relational BDs