Query rewriting

In the previous chapters, we saw how to saturate a fact base with a set of rules, applying them exhaustively so that any query evaluation automatically considers all consequences of those rules.

In this document, we explore an alternative strategy: query rewriting.

Instead of enriching the fact base, query rewriting expands the query itself using the rules. The result is a new query – or more precisely, a union of queries – that captures all ways the original question could be answered given the rule base.

This approach is widely used in ontology-based data access (OBDA) systems because:

- It avoids modifying or copying large fact bases.
- It can be combined with existing database engines by rewriting the query into SQL.
- In some cases, it guarantees termination where saturation would produce an infinite chase.

We will cover the intuition, the DLGP syntax examples, and how rewriting relates to saturation through equivalence properties.

All code examples shown here can be found in the accompanying Python file: ex4_query_rewriting.py.

Rewriting a query

Saturation is not the only way to take a rule base into account during evaluation. **Query rewriting** achieves the same goal, but instead of expanding the fact base, it transforms the query itself to incorporate the rules.

Let us consider now the rule `mortal(X) :- human(X).`, the query `?(X) :- mortal(X).`, and the factbase `human(socrates)`. Until now, to answer this query in the factbase while taking the rulebase into account, we used saturation (see ex3):

```
query.evaluate(factbase.saturate(rulebase))
```

What is rewriting

Query rewriting takes a different approach. Instead of expanding the fact base, it expands the query itself. To answer "find all mortals", the rewriting algorithm looks at the rules and deduces that *finding a human is enough to conclude that it's a mortal.*

The rewritten query therefore becomes:

> *"Find all X such that either X is mortal or X is human."*

This produces a union of conjunctive queries (UCQ). The algorithm enriches the original query in place, generating all equivalent formulations that take the rules into account.

Example:

```
from py4graal import *

rulebase = graal.create_rulebase("mortal(X) :- human(X).")
query = graal.create_query("?(X) :- mortal(X).")

query.rewrite(rulebase)
print(query)
```

Output:

```
?(X) :- mortal(X).
?(X) :- human(X).
```

Evaluation with a rewritten query

The key property of rewriting is that it preserves answers:

- `query.evaluate(factbase.saturate(rulebase))`
- `query.rewrite(rulebase).evaluate(factbase)` ...return exactly the same results.

Example:

```
factbase = graal.create_factbase("human(socrates). human(plato).
human(aristotle).")
rulebase = graal.create_rulebase("mortal(X) :- human(X).")
query = graal.create_query("?(X) :- mortal(X).")

query.rewrite(rulebase)
print(list(query.evaluate(factbase)))
```

Output:

```
[{'X': 'aristotle'}, {'X': 'socrates'}, {'X': 'plato'}]
```

Terminations of rewriting vs saturation

Rewriting and saturation don't always behave the same way regarding termination. Some rule bases make one process loop forever while the other halts.

Infinite saturation, finite rewriting

In ex3_rules_saturation, we saw that the rule base `parent(Y, X), human(Y) :- human(X).` can cause infinite saturation: as soon as there exists a human in the rulebase, that human generates a new parent, which generates another human, and so on.

However, query rewriting with the same rules always terminates (whatever the query), because each rewriting step with this rule proves (and erases) at least one atom, and replace it with the unique atom in the body. As a consequence, we only add to the rewritten query bodies of bounded size, and there can be only finitely many of them.

```
rulebase = graal.create_rulebase("parent(X, Y), human(Y) :- human(X).")
query = graal.create_query("?(X) :- parent(X, Y).")
print (query.rewrite(rulebase))
```

Output:

```
?(X) :- human(X).
?(X) :- parent(X, Y).
```

Infinite rewriting, finite saturation

Now consider the rule base `ancestor(X, Z) :- ancestor(X, Y), ancestor(Y, Z).` This is a **transitivity rule**: if X is an ancestor of Y and Y is an ancestor of Z, then X is an ancestor of Z.

- Saturation with this rule terminates on any finite fact base because the rule does not have any existential variable (it is a *Datalog* rule), and will never introduce new individuals on the fact base. Since finitely many atoms can be added on a finite set of individuals, saturation will terminate.
- Query rewriting, however, will endlessly generate new queries, each `ancestor(X, Y)` atom being rewritten as a pair of atoms, and thus always generating a conjunctive query of size n+1 from a conjunctive query of size n, as shown below.

```
?(X, Z) :- ancestor(X, Z).
?(X, Z) :- ancestor(X, Y), ancestor(Y, Z).
?(X, Z) :- ancestor(X, Y1), ancestor(Y1, Y2), ancestor(Y2, Z).
?(X, Z) :- ancestor(X, Y1), ancestor(Y1, Y2), ancestor(Y2, Y3), ancestor(Y3, Z).
```

The program below, however, shows that saturation terminates:

```
factbase = graal.create_factbase("human(socrates), ancestor(daedalus,
sphoroniscus), ancestor(sophroniscus, socrates).")
rulebase = graal.create_rulebase("ancestor(X, Z) :- ancestor(X, Y), ancestor(Y,
Z).")
print(factbase.saturate(rulebase))
```

Output:

```
ancestor(daedalus, socrates), ancestor(daedalus, sophroniscus),
ancestor(sophroniscus, socrates), human(socrates).
```