Three Query Language Formalisms

Dan Suciu

September 25, 2011

This is supplemental material for various database courses offered at the University of Washington. It is required reading material for CSE544 and CSEP544, and it is optional reading material for CSE344 and CSE444.

1 Introduction

We describe three formalisms, or languages, for expressing queries: (1) relational algebra, (2) non-recursive datalog with negation, and (3) relational calculus. We also discuss how to translate between these three language. The three formalisms capture different aspects of database queries: understanding them, and understanding the translation between them will strongly enhance your ability to master complex database queries, and to understand concepts in query optimization and query execution.

Before we start, let's talk about schemas and queries.

Schemas By the *schema* of a relation we mean its attributes. It turns out there are two ways to define the schema: in the *named perspective* each attribute has a name, and the schema is a *set of attribute names*. In the *unnamed* perspective attributes have no name, and we identify them by their position; the schema of a relation is a number, called *arity*. Here is the schema of `Actor`, `Casts`, `Movie` in the two formalisms, and how we refer to the `First Name`:

	Named Perspective	Unnamed Perspective
Schema:	`Actor(id, fname, lname)`	3
	`Casts(pid, mid)`	2
	`Movie(id, title, year)`	3
Attribute reference:	`Actor.fname`	`Actor`.2

In the named perspective the order of attributes doesn't matter, it's the same if we write `Actor(lname, id, fname)`. In the unnamed perspective we just say that the arity 3; attributes are identified by their position, and we need to remember which is which. To select the first names, we write `Actor`.2 (assuming `fname` is the second attribute).

Queries A *query* Q is a function that takes as input m finite relations $R_1, \ldots, R_m$, and returns as output a finite relation: both inputs and output are sets, we will not discuss bag semantics here. We write $Q(R_1, \ldots, R_m)$ to emphasize that Q is a function with inputs $R_1, \ldots, R_m$.

Keep in mind that all queries are *typed*: the schemas of both input relations and the output are fixed. For example, consider the following SQL query:

```
Q: select distinct a.fname, a.lname
   from Actor a, Casts c1, Movie m1, Casts c2, Movie m2
   where a.id = c1.pid and c1.mid = m1.id
     and a.id = c2.pid and c2.mid = m2.id
     and m1.year = 1910 and m2.year = 1940;
```

Then the query takes three input relations, `Actor`, `Casts`, `Movie`, and returns an output relation of arity 2. We assume that the schemas of the three relations to be given, as above; the schema of the output relation is also fixed, namely (`fname`, `lname`).

2 Relational Algebra

Relational Algebra (RA) is used by query optimizers as an intermediate language for query optimization, and also for query execution. We cover RA in every database class offered at UW. RA is meant to be executed by the system, not for expressing queries: users almost never have to write queries in directly RA, they write queries in some other language (usually

SQL) and the system translates these to RA. However, some recent, modern query languages, like Pig Latin, look quite a bit like RA, so you should feel familiar writing queries directly in RA if needed.

RA has five operators:

Selection $\sigma_C(R)$, where C is a Boolean condition.

Projection $\Pi_A(R)$, where A is a list of attributes.

Join $R_1 \bowtie_C R_2$, where C is a Boolean condition.

Union $R_1 \cup R_2$.

Difference $R_1 - R_2$.

If we use the unnamed perspective, then that's all. If we use the named perspective, then we need one more operation, *renaming* $\rho_A(R)$, where A is a list of renamed attributes. You should know very well what these operators mean: if not, please review the lecture notes, or read in textbook.

An *RA query* is an expression consisting of these operators, applied to some base relations $R_1, R_2, \ldots$ We always insist that the expressions are correctly typed, for example we cannot write $\texttt{Actor} \cup \texttt{Casts}$ because the two relations have different arities; we cannot write $\Pi_{\texttt{id}}(\texttt{Actor}) \cup \Pi_{\texttt{pid}}(\texttt{Cast})$ either (why not ?).

For an example example, the SQL query Q from Section 1 is translated into the following RA expressions, in the unnamed and in the named perspective respectively:

$$\begin{aligned} Q = \Pi_{2,3}(\sigma_{8='1910' \wedge 13='1940'}((\texttt{Actor} \bowtie_{1=1} (\texttt{Casts} \bowtie_{2=1} \texttt{Movie})) \\ \bowtie_{1=1} (\texttt{Casts} \bowtie_{2=1} \texttt{Movie})) \end{aligned}$$

$$\begin{aligned} Q = \Pi_{\texttt{fname},\texttt{lname}}(\sigma_{y1='1910' \wedge y2='1940'}((\texttt{Actor} \bowtie_{\texttt{id}=\texttt{pid}} (\texttt{Casts} \bowtie_{\texttt{mid}=\texttt{id}} \rho_{y1=\texttt{year}}(\texttt{Movie}))) \\ \bowtie_{\texttt{id}=\texttt{pid}} (\texttt{Casts} \bowtie_{\texttt{mid}=\texttt{id}} \rho_{y2=\texttt{year}}(\texttt{Movie}))) \end{aligned}$$

Theoreticians prefer the concise notation of the unnamed perspective. In practice, the named perspective is more user friendly, but we must rename attributes to disambiguate. Make sure you understand in detail the two expressions above; make sure you understand how renaming was used to disambiguate between attribute names.

Sometimes, we abuse the notation and we drop the renaming altogether, using tuples variables variables instead, as in:

$$Q = \Pi_{\texttt{fname},\texttt{lname}}(\sigma_{\texttt{m1.year}='1910' \wedge \texttt{m2.year}='1940'}((\texttt{Actor} \bowtie_{\texttt{id}=\texttt{pid}} (\texttt{Casts} \bowtie_{\texttt{mid}=\texttt{id}} \texttt{Movie m1})) \\ \bowtie_{\texttt{id}=\texttt{pid}} (\texttt{Casts} \bowtie_{\texttt{mid}=\texttt{id}} \texttt{Movie m2}))$$

Monotone Queries Selection, project, join, and union are monotone operators. Every query written using only these four operators is monotone. Difference is the only non-monotone operator in RA.

3 Non-Recursive Datalog With Negation

Datalog is the simplest, and most elegant formalisms for queries. Historically, it was introduced in order to add recursion to query languages, and it was designed as a fragment of Prolog, with a bottom-up evaluation strategy instead of top-down. While some commercial datalog implementations exists, you probably won't encounter them; instead, you will often find yourself using datalog to sketch complex queries first, before converting them to SQL. Think about datalog as SQL with a nicer syntax. We do not discuss datalog in every database course; if we didn't discuss it in your class, then read this material, it should be sufficient to learn datalog.

Datalog uses *only* the unnamed perspective.

3.1 Non-Recursive Datalog

Let's see examples:

```
Q1(y) :- Movie(x,y,1940)
```

This query finds the titles of all movies made in 1940: more precisely, it retrieves all tuples of the form `(x,y,1940))` in `Movie` (hence, the year must be 1940), and returns `y` (the title). Such a query is called a *conjunctive query*, or a *datalog rule*.

Consider now the following datalog rule:

```
Q2(f,l) :- Actor(z,f,l), Casts(z,x), Movie(x,y,1940)
```

This returns the list of first and last names of all actors who acted in movies made in 1940. Notice that when a variable occurs twice, then it must be the same value. For example, `z` occurs twice: this says that `Actor.id` id must be equal to the `Casts.pid`; in other words, it joins `Actor` and `Casts`. Similarly `x` joins `Casts` and `Movie`. If you'd like, you can write these equalities explicitly, as in the following datalog rule, which computes exactly the same query:

```
Q2(f,l) :- Actor(z1,f,l), Casts(z2,x1), Movie(x2,y,1940), z1=z2, x1=x2
```

Finally, consider the query Q in Section 1. Here is how it looks in datalog:

```
Q3(f,l) :- Actor(z,f,l), Casts(z,x1), Movie(x1,y1,1910),
                         Casts(z,x2), Movie(x2,y2,1940)
```

This looks much simpler than SQL !

So far, every query we wrote consisted of a single datalog rule. In datalog, a program consists of several rules. Let's see an example: the datalog program below computes all actors whose Bacon number is 1 or 2:

```
B0(x) :- Actor(x,'Kevin', 'Bacon')
B1(x) :- Actor(x,f,l), Casts(x,z), Casts(y,z), B0(y)
B2(x) :- Actor(x,f,l), Casts(x,z), Casts(y,z), B1(y)
Q4(x) :- B1(x)
Q4(x) :- B2(x)
```

Note that here we use several rules. The first computes the relation `B0(x)`, which will contain the `id` of Kevin Bacon: thus, `B0` contains actors with Bacon number 0. The second rule computes `B1(x)`, the actors with Bacon number 1. The third rule computes `B2(x)`, the actors with Bacon number 2: notice how it refers to `B1` (because an actor has Bacon number 2 if she acted in a movie with an actor with Bacon number 1). Now look carefully at the last two rules, which define the query's output `Q4`: every `x` in `B1` is copied to `Q4`, then every `x` in `B2` is copied to `Q4`. Thus, `Q4` is the union of `B1` and `B2`.

Congratulations: you have just learned datalog!

3.2 Terminology

A query in datalog is called a *program*, and is a sequence of *rules*. Each rule has the form:

$$H(\mathbf{x}) : -P_1, \ldots, P_m \tag{1}$$

where H is a relational symbol, $\mathbf{x}$ is a list of variables, and $P_1, \ldots, P_m$ are atomic predicates. A predicate is either a *relational predicate*, meaning $R(t_1, \ldots, t_k)$ where $t_1, \ldots, t_k$ are variables or constants, or a *interpreted predicate*, like $x = y$ or $2 * x > y + z$.

$H(\mathbf{x})$ the rule's *head*, and $P_1, \ldots, P_m$ is the rule's *body*. This must be a new name, which is not part of the database relations $R_1, \ldots, R_m$: think of H as the name of an output relation. The input relations $R_1, \ldots, R_m$ are called *extensional database relations*, or EDB; all output relations (occurrindin some rule's head) are called *intentional database relations*, or IDB. In all three examples in Subsection 3.1, the EDBs are `Actor`, `Casts`, `Movie`, and the IDBs are `Q1, Q2, Q3, B1, B2, Q4`.

3.3 No Recursion

In the rule Equation 1, the IDB H is defined in terms of $P_1, \ldots, P_m$, which may be either EDBs or IDBs. The datalog program is called *recursive* if some IDB is defined recursively, either directly, or indirectly. One of the main motivation of datalog was to express recursion: datalog programs can be recursive. However, in this tutorial, we will only discuss non-recurisve datalog programs, hence we will assume our datalog programs to be non-recursive.

But before we restrict to non-recursive programs, let's have a short digression into recursion. To seen an example of a recursive program, consider a graph defined by a binary relation, $E(x, y)$. That is, if two nodes a, b are connected by an edge in the graph, then E contains the tuple (a, b). The following recursive datalog program computes the transitive closure of the graph:

$$\begin{aligned} T(x, y) &: -E(x, y) \\ T(x, z) &: -E(x, y), T(y, z) \end{aligned}$$

Here T is an IDB that is defined recursively.

In the 1980's SQL could not express recursive queries, and promoters of datalog were arguing that such an extension was needed. However, the industry was slow to embrace recursion: customers were using SQL in OLTP queries, and they didn't need recursion: they needed better indices, better join algorithms, better optimizations, and better transactions processing. So this is what the industry delivered; recursion was not added to SQL until many years later, and even today the support of recursion in SQL engines is dismal, and the syntax of recursive SQL queries is horrible!.

However, today there is a renewed interest in supporting recursive queries, because of the advent of *big data*. This is a vague term that usually means massive data, complex queries, and parallelism. Many of these complex "queries" require recursion: analyzing large social networks, computing page rank, computing connected components in large graphs. This is causing today a renewed interest recursive queries.

3.4 Adding Negation

A datalog program, recursive or not, can express only monotone queries. To express non-monotone queries, we need to add negation. As before, we start with some examples.

The following query returns all movies from 1994 where Kevin Bacon did not act:

```
Q5(x) :- Movie(x,y,1994), Actor(z,'Kevin','Bacon'), not Casts(z,x)
```

Note the use of `not`: the query says that the predicate `Casts(z,x)` should be false. Seems easy, but consider this. Suppose there are multiple actors with the name Kevin Bacon: thus, the variable `z` can be bound to several actor id's. Then, what should the query return? The options are (a) it returns every movie `x` such that at least one of the Kevin Bacon's did not act in `x`; or (b) it returns every movie `x` such that none of the Kevin Bacon's acted in `x`. Which one is it? (If you can't figure out, then read on...)

For another example, consider the query below. It returns all actors whose Bacon number is ≥ 2:

```
B0(x) :- Actor(x,'Kevin', 'Bacon')
B1(x) :- Actor(x,f,l), Casts(x,z), Casts(y,z), B0(y)
Q6(x) :- Actor(x,f,l), not B1(x), not B0(x)
```

B0 and B1 are as before: they consists of all actors with Bacon number 0 or 1 respectively. The query simply returns the actors that are neither in B0 nor in B1.

Now the formal definition. A *datalog rule with negation* is a rule as given by Equation 1, where each P_i is either a relational predicate, the negation of a relational predicate, or an interpreted predicate. A *non-recursive datalog program with negation* is a list of datalog rules with negation, which is non-recursive as defined earlier.

3.5 Safe Datalog

The following are "unsafe" datalog rules:

```
U1(x,y) :- Movie(x,z,1994)
U2(x)   :- Movie(x,z,1994), not Casts(u,x)
```

In the first rule, who is `y` ? It is undefined, and the query can't return it: we say that the first rule is unsafe, and will simply forbid it from datalog. The second rule is also unsafe, but it is more subtle to see that. Here `u` is used in the negation, but it is not bound anywhere else, and there are two interpretations for `not Casts(u,x)`: (a) there exists `u` such that `Casts(u,x)` is not true, or (b) it is not true that there exists `u` such that `Casts(u,x)`. We don't want such ambiguities, so this rule is also unsafe, and hence forbidden.

A datalog rule is *safe* if every variable appears in a positive relational atom. A datalog program is safe if all its rules are safe. Every datalog program must be safe.

3.6 Datalog v.s. SQL

Non-recursive datalog with negation is just another syntax for SQL. In other words, you should feel comfortable moving back and forth between these two languages. We illustrate this idea at the end of this tutorial.

3.7 Translation 1: From Non-recursive datalog with Negation to Relational Algebra

We can now describe our first translation: from non-recurisve datalog with negation, to RA.

We start with a single rule:

$$H(\mathbf{x}) :- P_1, \ldots, P_m, \ \texttt{not} \ N_1, \ldots, \texttt{not} \ N_k$$

Let $\mathbf{y}$ be all variables that appear in $P_1, \ldots, P_m$. We translate the rule into the following RA expression:

$$H = \Pi_{\mathbf{x}}[\Pi_{\mathbf{y}}(P_1 \bowtie \ldots \bowtie P_m) - \Pi_{\mathbf{y}}(P_1 \bowtie \ldots \bowtie P_m \bowtie N_1) - \ldots - \Pi_{\mathbf{y}}(P_1 \bowtie \ldots \bowtie P_m \bowtie N_k)]$$

For example, consider the rule:

$$H(x, y) :- R(x, u, v), S(y, u), \ \texttt{not} \ T(x, v)$$

It becomes:

$$H = \Pi_{1,4}[\Pi_{1,2,3,4}(R \bowtie_{2=2} S) - \Pi_{1,2,3,4}((R \bowtie_{2=2} S) \bowtie_{1=1 \wedge 3=2} T)$$

Why can't we subtract T directly from $R \bowtie S$, writing something like $\Pi(R \bowtie S - T)$? In other words, why do we need to join T with R and S before subtracting ? Make sure you understand why.

Once you understood how to translate one rule, translating an entire program is easy. Let $H_1, H_2, \ldots, H_m$ be all the IDB predicates occurring in the datalog rule, in this order. For each predicates H_i, construct a relational algebra expression for each rule defining H_i, as shown above, then take their union: note that each such expression may refer to $H_1, \ldots, H_{i-1}$, for which we simply substitute with their definitions. **Exercise:** translate `Q4` to RA.

4 The Relational Calculus

The Relational Calculus (RC) is the same as First Order Logic, and the same as Predicate Logic. We do not discuss RC in our database classes at UW (except in 544), but mention its salient features, such as universal and existential quantifiers: the reason why you should know RC is precisely to master the use of these quantifiers. However, even though we don't cover it in the database classes, you already know it! You have encountered it in Discrete Mathematics, under the name Predicate Logic: this is the same as RC. We review RC below, giving a complete definition of RC: however, to refresh your memory about all its subtleties, please review it from your favorite textbook in Discrete Mathematics.

4.1 Language Definition

A *Relational Predicate* P is a formula given by the following grammar:

$$P ::= \texttt{A} \mid P \wedge P \mid P \vee P \mid \neg P \mid \exists x.P \mid \forall x.P \tag{2}$$

Here `A` stands for an atomic predicate, which is either a relational predicate or an interpreted predicate, see Subsection 3.2. The other connectives should be familiar from Discrete Mathematics: and, or, negation, existential quantifier, and universal quantifier.

A *Query* in the Relational Calculus is an expression of the form:

$$Q(\mathbf{x}) \equiv P$$

where P is an expression given by the grammar in Equation 2 and $\mathbf{x}$ are its free variables[1]

Don't panic yet, the language is actually quite simple. Let's look at some examples.

Find all movies made in 1940:

$$\texttt{Q7}(y) \equiv \exists x.\texttt{Movie}(x, y, 1940)$$

Note how we manage variables: x is existentially quantified, hence the only free variable is y, so we write it as head variable in `Q7`.

Find the names of all actors who acted in movies from 1940:

$$\texttt{Q8}(f, l) \equiv \exists x.\exists y.\exists z.\exists y.\texttt{Actor}(z, f, l) \wedge \texttt{Casts}(z, x) \wedge \texttt{Movie}(x, y, 1940)$$

[1]The set of free variables in P, denoted $\mathit{Vars}(P)$, is defined formally by:

$$\begin{aligned}
&\mathit{Vars}(R(t_1, \ldots, t_k)) = \{t_i \mid t_i = \text{ a variable }\}\\
&\mathit{Vars}(P_1 \wedge P_2) = \mathit{Vars}(P_1 \vee P_2) = \mathit{Vars}(P_1) \cup \mathit{Vars}(P_2)\\
&\mathit{Vars}(\neg P) = \mathit{Vars}(P)\\
&\mathit{Vars}(\exists x.P) = \mathit{Vars}(\forall x.P) = \mathit{Vars}(P) - \{x\}
\end{aligned}$$

Exercise Write the SQL query from Section 1 in RC. What kind of quantifiers did you need, existential or universal ?

The power of universal quantifiers The true power of RC comes from the use of universal quantifiers. Consider for example the following query: find all actors who acted only in 1940:

$$\texttt{Q9}(f,l) \equiv \exists x.(\texttt{Actor}(z,f,l) \wedge \forall y.\forall v.(\texttt{Casts}(z,x) \wedge \texttt{Movie}(x,y,v) \Rightarrow v = 1940))$$

Here $\Rightarrow$ is logical implication (you probably know what it means; if not, then read below). To appreciate RC, try to write this query in SQL, or in RA: it is possible, but much harder. This is why we need to know RC, to be able to express easily queries with universal quantifiers. Of course, to actually run these queries we need to write them in SQL first. We discuss the translation from RC to non-recursive datalog with negation below: getting from there to SQL is a breeze.

Exercise Write the following queries in the relaional calculus.

- Find actors who played in every year of their careers. That is, return an actor if she played in (say) exactly in 1950, 51, 52, 53, 54, but don't return that actor is she played in 1950, 52, 53, 54 (there is a gap at 51). In this query you may use interpreted predicates like $x < y$ and $x + 1 = y$.
- Find actors that played only in movies where Kevin Bacon also played.
- Find all movies that have only actors that acted only in movies where Kevin Bacon also played.

Review of predicate logic At this point, it is useful to recall some very basic stuff from predicate logic:

- $P_1 \Rightarrow P_2$ is a notation for $\neg P_1 \vee P_2$. We will freely use $\Rightarrow$ in RC.
- De Morgan: $\neg(P_1 \wedge P_2) \equiv (\neg P_1) \vee (\neg P_2)$ and $\neg(P_1 \vee P_2) \equiv (\neg P_1) \wedge (\neg P_2)$.
- Same quantifiers commute: $\exists x.\exists y.P \equiv \exists y.\exists x.P$ and $\forall x.\forall y.P \equiv \forall y.\forall x.P$.
- Different quantifiers do not commute in general: $\exists x.\forall y.P \not\equiv \forall y.\exists x.P$.
- De Morgan for quantifiers: $\neg(\exists x.P) \equiv \forall x.(\neg P)$ and $\neg(\forall x.P) \equiv \exists x.(\neg P)$.

RC v.s. datalog Finally, let's discuss the relationship between datalog and RC. The key fact to remember is that a datalog rule has only existential quantifiers: every variable in the body that is not a head variable is existentially quantified. For example, recall the query `Q5`:

```
Q5(x) :- Movie(x,y,1994), Actor(z,'Kevin','Bacon'), not Casts(z,x)
```

This is equivalent to:

$$\texttt{Q5}(x) \equiv \exists y.\exists z.(\texttt{Movie}(x, y, 1944) \wedge \texttt{Actor}(z, \texttt{'Kevin', 'Bacon'}) \wedge \neg\texttt{Casts}(z, x))$$

Now the answer to our earlier question becomes obvious: it is (a).

4.2 The Drinkers-Bars-Beers Example

This is a famous example, first used by Ullman. Consider the relational schema:

```
Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)
```

For example, the query below lists all bars that serve some beer that none of their patrons like:

$$\texttt{Q10}(x) \equiv \exists y.\texttt{Serves}(x, y) \wedge \forall z.(\texttt{Freqents}(z, x) \Rightarrow \neg\texttt{Likes}(z, y))$$

Write each query below in RC.

- Find all drinkers that frequent some bar that serves some beer that they like.
- Find all drinkers that frequent only bars that serve some beer that they like.
- Find all drinkers that frequent some bar that serves only beer that they like.
- Find all drinkers that frequent some bar that serves some beer that they don't like.

- Find all drinkers that ferquent some bar that serves only beer that they don't like.
- Find all drinkers that frequent only bars that serve some beer that they don't like.
- Find all drinkers that frequent only bars that serve only beer that they don't like.

Make sure you write *simple* RC queries for each question; the idea is that one can go from English to RC almost automatically.

4.3 Domain Independent RC

As with datalog, we can write in RC queries that make no sense. Consider the following query:

$$F(t) \equiv \forall y.\exists x.\texttt{Movie}(x, t, y)$$

The query returns the titles that were made in *every* year. It will return no answers at all on the IMDB dataset, since no movie title was produced in every single year. One movie that comes close is "Reunion", which is a title given to movies made in 40 different years, 1932, 1936, 1954, 1955, ..., 2008, and 2009. But let's imagine that "Reunion" *were* made every single year, so that the query should return it. But what does *every year* even mean ? Should it mean every year from, say, 1900 to 2011 ? Or every year from 1890 to 2015 ? Thinking about it, you realize that the answer to the query depends not only on what is in the table `Movie`, but also on the domain for the year variable y. If "every y" means "every y in $[1900, 2011]$" then the query has one meaning; but if "every y" means "every y in $[0, 2020]$" then the query has a different meaning. We say that the query `F` is *domain dependent*. A domain dependent query makes no sense in databases, since we are interested only in querying the tables in the database, not the domain.

Notice that in Predicate Logic we don't hesitate writing sentences that depend on the domain. Consider $\forall x.\exists y.y < x$: true or false ? Well, it depends on the domain. Over the natural numbers the sentence is false (it fails for $x = 0$); over integers, or rationals, or reals, the sentence is true.

A query in RC must be *domain independent*, i.e. its answer must depend only on the relations in the database, not on the domain. This is the same as safety in datalog rules. However, achieving domain independence in RC is trickier than safety in datalog: we won't say how to do that, but will show how *not* to write queries. Below is a list of domain dependent queries: don't write queries like that, but make sure you understand why they are domain dependent.

$$\begin{aligned}
F_1(x,y) \equiv& \exists z.\texttt{Movie}(x,z,1994) \wedge y \neq z \\
F_2(x) \equiv& \exists z.\exists u.(\texttt{Movie}(x,z,1994) \wedge \neg\texttt{Casts}(u,x)) \\
F_2(u,v) \equiv& \exists f.\exists l.\texttt{Actor}(u,f,l) \vee \exists t.\exists y.\texttt{Movie}(v,t,y)
\end{aligned}$$

We will denote with $D(z)$ the active domain of a database. For example, in the case of the Drinkers/Bars/Beers example, the active domain D consists of all drinkers, all bars, and all beers in the database, and can be computed by the following datalog program:

```
D(z) :- Frequents(z,x)
D(z) :- Frequents(x,z)
D(z) :- Likes(z,x)
D(z) :- Likes(x,z)
D(z) :- Serves(z,x)
D(z) :- Serves(x,z)
```

4.4 Translation 2: from RC to Non-recursive datalog with Negation

Consider a query given by Equation 2. To translate it into non-recursive datalog with negation, start by writing all its subexpressions, call them $P_1, P_2, \ldots, P_m$. List them in an order such that if P_i is a subexpression of P_j then $i < j$, and let $\mathbf{x}_1, \ldots, \mathbf{x}_m$ be their free variables. For each of them, define the corresponding query, denoted $H_1, \ldots, H_m$:

$$\begin{aligned}
H_1(\mathbf{x}_1) \equiv& P_1 \\
& \cdots \\
H_m(\mathbf{x}_m) \equiv& P_m
\end{aligned}$$

We make two assumptions about the formulas $P_1, \ldots, P_m$.

- First, we assume that there are no universal quantifiers. Indeed, this can be achieved using de Morgan's laws: $\forall z.P_i \equiv \neg\exists z.\neg P_i$.
- Second, we assume that every negation occurs in a context $P_i \wedge \neg P_j$. Furthermore, we assume that all free variables of P_j occur as free variables in P_i (i.e. $\mathbf{x}_j \subseteq \mathbf{x}_i$). This is always possible. For most practical queries this assumption already holds. If this assumption doesn't hold, then we can always replace any subformula $\neg P_j$ with $D(z_1) \wedge \ldots \wedge D(z_k) \wedge \neg P_j$, where $z_1, \ldots, z_k$ are the free variables in P_j and D is the formula computing the active domain of the database: since the query is domain independent, this change will not affect its semantics.

Next, for each $i = 1, m$, we will rewrite H_i in terms of datalog rules, instead of RC formulas. This is done recursively on the structure of P_i, according to the the grammar rules in Equation 2:

Atomic Predicate If P_i is a relational atomic predicate $P(t_1, \ldots, t_k)$ then the rule for H_i is:

$$H_i(\mathbf{x}_i) :- P(t_1, \ldots, t_k)$$

If P_i is an interpreted predicate, C with free variables $z_1, \ldots, z_k$ then we write:

$$H_i(\mathbf{x}_i) :- D(z_1), \ldots, D(z_k), C$$

(In practice this latter rule can often be simplified.) Clearly, both rules are safe.

Conjunction If $P_i = P_j \wedge P_k$ then write:

$$H_i(\mathbf{x}_i) :- H_j(\mathbf{x}_j), H_k(\mathbf{x}_k)$$

The rule is safe because $\mathbf{x}_i = \mathbf{x}_j \cup \mathbf{x}_k$ (why ?).

Disjunction If $P_i = P_j \vee P_k$ then define H_i via two rules:

$$H_i(\mathbf{x}_i) :- H_j(\mathbf{x}_j)$$
$$H_i(\mathbf{x}_i) :- H_k(\mathbf{x}_k)$$

The rule is safe because $\mathbf{x}_i = \mathbf{x}_j = \mathbf{x}_k$. (Why ?)

Negation By our assumption negation occurs in the following context: $P_i = P_k \wedge \neg P_j$. Then we write:

$$H_i(\mathbf{x}_i) : -H_k(\mathbf{x}_k), \neg H_j(\mathbf{x}_j)$$

Existential quantifier If $P_i = \exists z.P_j$ then $\mathbf{x}_i = \mathbf{x}_j - \{z\}$ and we write:

$$H_i(\mathbf{x}_i) : -H_j(\mathbf{x}_j)$$

4.5 Example of a Translation

Consider the following query:

$$\texttt{Q11}(x) \equiv \exists y.\texttt{Likes}(x, y) \wedge \forall z.(\texttt{Serves}(z, y) \Rightarrow \texttt{Frequents}(x, z))$$

We first rewrite `Q11` by applying de Morgan's law to the universal quantifier, then to $\Rightarrow$ (recall that $\neg(A \Rightarrow B) \equiv A \wedge \neg B$):

$$\texttt{Q11}(x) = \exists y.\texttt{Likes}(x, y) \wedge \neg\exists z.(\texttt{Serves}(z, y) \wedge \neg\texttt{Frequents}(x, z))$$

The first $\neg$ satisfies our assumption: it occurs in the context $\texttt{Likes}(x, y) \wedge \neg(\ldots)$ and the free variables in $(\ldots)$ are x and y, so they both occur in the non-negated formula, $\texttt{Likes}(x, y)$.

The second $\neg$ does not satisfy our assumption: it occurs in the context $\texttt{Serves}(z, y) \wedge \neg\texttt{Frequents}(x, z)$, so the negated predicate has the extra variable x. However, note that the outer predicate $\texttt{Likes}(x, y)$ binds x, so the query is equivalent to the following:

$$\texttt{Q11}(x) \equiv \exists y.\texttt{Likes}(x, y) \wedge \neg\exists z.(\texttt{Likes}(x, y) \wedge \texttt{Serves}(z, y) \wedge \neg\texttt{Frequents}(x, z))$$

Now the query satisfies our assumptions. There are 7 subformulas, and for each we define a new IDB:

$$\begin{aligned}
\texttt{H}_1(x, z) &\equiv \texttt{Frequents}(x, z)\\
\texttt{H}_2(y, z) &\equiv \texttt{Serves}(z, y)\\
\texttt{H}_3(x, y) &\equiv \texttt{Likes}(x, y)\\
\texttt{H}_4(x, y, z) &\equiv \texttt{Likes}(x, y) \wedge \texttt{Serves}(z, y) \wedge \neg\texttt{Frequents}(x, z)\\
\texttt{H}_5(x, y) &\equiv \exists z.(\texttt{Likes}(x, y) \wedge \texttt{Serves}(z, y) \wedge \neg\texttt{Frequents}(x, z))\\
\texttt{H}_6(x, y) &\equiv \texttt{Likes}(x, y) \wedge \neg\exists z.(\texttt{Likes}(x, y) \wedge \texttt{Serves}(z, y) \wedge \neg\texttt{Frequents}(x, z))\\
\texttt{H}_7(x) &\equiv \exists y.\texttt{Likes}(x, y) \wedge \neg\exists z.(\texttt{Likes}(x, y) \wedge \texttt{Serves}(z, y) \wedge \neg\texttt{Frequents}(x, z))
\end{aligned}$$

(We took one shortcut: the formula $\mathtt{H}_4$ should be split into two: first $\mathtt{H}'_4(x, y, z) = \mathtt{Likes}(x, y) \wedge \mathtt{Serves}(z, y)$, then $\mathtt{H}_4(x, y, z) = \mathtt{H}'_4(x, y, z) \wedge \neg\mathtt{Frequents}(x, z)$. Instead, we lumped them together into a single formula, for simplicity.)

Now the datalog program follows immediately, by following the method described above:

$$
\begin{aligned}
\mathtt{H}_1(x, z) &: -\mathtt{Frequents}(x, z)\\
\mathtt{H}_2(y, z) &: -\mathtt{Serves}(z, y)\\
\mathtt{H}_3(x, y) &: -\mathtt{Likes}(x, y)\\
\mathtt{H}_4(x, y, z) &: -\mathtt{H}_3(x, y), \mathtt{H}_2(z, y),\ \mathtt{not}\ \mathtt{H}_1(x, z)\\
\mathtt{H}_5(x, y) &: -\mathtt{H}_4(x, y, z)\\
\mathtt{H}_6(x, y) &: -\mathtt{H}_3(x, y),\ \mathtt{not}\ \mathtt{H}_5(x, y)\\
\mathtt{H}_7(x) &: -\mathtt{H}_6(x, y)
\end{aligned}
$$

We can simplify this, by removing the IDB's $\mathtt{H}_4$ and $\mathtt{H}_6$, writing:

$$
\begin{aligned}
\mathtt{H}_1(x, z) &: -\mathtt{Frequents}(x, z)\\
\mathtt{H}_2(y, z) &: -\mathtt{Serves}(z, y)\\
\mathtt{H}_3(x, y) &: -\mathtt{Likes}(x, y)\\
\mathtt{H}_5(x, y) &: -\mathtt{H}_3(x, y), \mathtt{H}_2(z, y),\ \mathtt{not}\ \mathtt{H}_1(x, z)\\
\mathtt{H}_7(x) &: -\mathtt{H}_3(x, y),\ \mathtt{not}\ \mathtt{H}_5(x, y)
\end{aligned}
$$

The SQL expression follows easily from the datalog program. First, write it ad literam:

```
H1(x,z): select distinct drinker as x, bar as z from Frequents
H2(y,z): select distinct bar as z, beer as y from Serves
H3(x,y): select distinct drinker as x, bar as y from Likes
H5(x,y,z): select distinct H3.x, H3.y from H2, H3
           where H2.y=H3.y
             and not exists (select distinct * from H1
                             where H1.x=H3.x and H1.z=H2.z)
H7(x):  select distinct H3.x  from H3
        where not exists (select distinct * from H5
                          where H5.x=H3.x and H5.y=H3.y)
```

then substitute each subquery and flatten the resulting query as much as possible, arriving at:

```
Q11: select distinct l.x from Likes l
     where not exists
         (select distinct * from Serves s, Likes l2
          where s.beer = l2.beer
            and not exists (select distinct * from Frequents f
                            where f.drinker = l2.drinker
                              and f.bar = s.bar))
```

The astute reader may notice that (a) we don't need to eliminate duplicates before checking if a subquery is non-empty, and (b) `Likes l2` is actually not needed. The query can be further simplified, and becomes:

```
Q11: select distinct l.x from Likes l
     where not exists
         (select *  from Serves s
          where s.beer = l.beer
            and not exists (select * from Frequents f
                            where f.drinker = l.drinker
                              and f.bar = s.bar))
```

4.6 Translation 3: from RA to RC

Finally, we close the loop by showing how to translate from RA to RC. This is very easy, but it is rarely ever needed.

Selection $\sigma_C(R)$ is translated into $P \wedge C$, where P is the translation of R.

Projection $\Pi_A(R)$ is translated into $\exists z_1 \ldots \exists z_m.P$, where P is the translation of R, and $z_1, \ldots, z_m$ are its free variables that are not included in the list of attributes A.

Join $R_1 \bowtie_C R_2$ is translated into $P_1 \wedge P_2 \wedge C$, where P_1, P_2 are the translations of R_1, R_2.

Union $R_1 \cup R_2$ is translated into $P_1 \vee P_2$.

Difference $R_1 - R_2$ is translated into $P_1 \wedge \neg P_2$.