

Three Query Language Formalisms

Dan Suciu

September 2011

Updated: January, 2012

This is supplemental material for various database courses offered at the University of Washington.

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 (<i>id</i> , <i>fname</i> , <i>lname</i>) Casts (<i>pid</i> , <i>mid</i>) Movie (<i>id</i> , <i>title</i> , <i>year</i>)	arity = 3 arity = 2 arity = 3
Attribute reference:	Actor.fname	Actor.2

In the named perspective the order of attributes doesn't matter. If we write **Actor**(*lname*, *id*, *fname*) or **Actor**(*lname*, *fname*, *id*), it's the same thing. In the unnamed perspective the order matters, because in the schema 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-name, we write **Actor.2** (assuming **fname** is the second attribute).

Queries A *query* Q is a function that takes as input some finite relations R_1, \dots, R_m , and returns as output one finite relation R . We do not discuss bag semantics here, so both inputs and the output are sets. We write $Q(R_1, \dots, R_m)$ to emphasize that Q is a function with inputs R_1, \dots, 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 class. 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, \dots . We always insist that the expressions are correctly typed, for example we cannot write $\mathbf{Actor} \cup \mathbf{Casts}$ because the two relations have different arities; we cannot write $\Pi_{\text{id}}(\mathbf{Actor}) \cup \Pi_{\text{pid}}(\mathbf{Cast})$ either (why not?).

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

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

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

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:

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

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 for recursive queries, but its simplicity makes it appealing today even for non-recursive queries. Think of non-recursive datalog as SQL with an elegant syntax. A few commercial datalog implementations exists, and several free implementations, but we will not use any of them in class; if you'd like to try out one, I recommend DLV, <http://www.dbai.tuwien.ac.at/proj/dlv/>. Since we cover datalog only briefly in class, read this material carefully. It should be sufficient to learn datalog.

Datalog uses *only* the unnamed perspective.

3.1 Non-Recursive Datalog

Let's see examples:

$Q1(y) :- \text{Movie}(x,y,1940)$

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* must be equal to *Casts.pid*, because the same variable *z* occurs in both positions. In other words, *z* 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 ???. 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)
```

So, you get the idea: one datalog rule is much like a **SELECT-FROM-WHERE** SQL query, only it looks much simpler than SQL!

Now lets discuss multiple datalog rules. Let's see the program below, which 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!

Exercise What do each of the datalog queries *E1*, *E2*, *E3*, *E4* compute?

$E1(x) :- R1(x), R2(x)$

$E2(x) :- R1(x)$

$E2(x) :- R2(x)$

$E3(x,y) :- R1(x), R2(y)$

$E4(x) :- R1(x), R2(y), x \neq y$

Exercise Consider a graph given as a binary relation $E(x,y)$. What does P5 compute below?

$P1(x,y) = E(x,z), E(z,y)$

$P2(x,y) = P1(x,z), P1(z,y)$

$P3(x,y) = P2(x,z), P2(z,y)$

$P4(x,y) = P3(x,z), P3(z,y)$

$P5(x,y) = P4(x,z), P4(z,y)$

3.2 Terminology

A *program* is a sequence of *rules*. Each rule is:

$$H(\mathbf{x}) : -P_1, \dots, P_m \quad (1)$$

where H is a relational symbol, \mathbf{x} is a list of variables, and P_1, \dots, P_m are atomic predicates. Each predicate is either a *relational predicate*, $R(t_1, \dots, t_k)$ where t_1, \dots, t_k are variables or constants, or an *interpreted predicate*, e.g. $x = y$ or $2 * x > y + z$. $H(\mathbf{x})$ the rule's *head*, and P_1, \dots, P_m is the rule's *body*.

The input relations R_1, \dots, R_m are called *extensional database relations*, or EDB. The output relations (those occurring in some rule's head) are called *intentional database relations*, or IDB. In our examples, the EDBs are Actor, Casts, Movie, and the IDBs are Q1, Q2, Q3, B1, B2, Q4.

3.3 No Recursion

In the rule ??, the IDB H is defined in terms of P_1, \dots, P_m . The datalog program is called *recursive* if some IDB is defined recursively in terms of itself, either directly, or indirectly. The simplest example of a recursive program

one that computes the transitive closure of a graph given by a binary relation, $E(x, y)$. Here it is:

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

The EDB is E and the IDB is T . While datalog was introduced precisely in order to express recursive queries, we will not discuss recursive queries in class; we consider only non-recursive datalog.

Modern SQL implementations also support recursive queries, but in rather limited fashion, and with a much more ugly syntax than datalog. The reason is simply that there is very little demand for recursion in commercial database applications.

However, modern data analytics, sometimes called *big data*, often do require recursion, hence the renewed interest in datalog today. For example, computing the friends-of-friends relationships in a graph, Google's page-rank algorithm, computing the *centrality score* in a social network, are all 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,u,'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. There are many actors named Bacon, thus **z** may be bound to several actor id's. What exactly should the query return? There are two options (a) return every movie **x** where at least one of the Bacon's did not act; or (b) it returns every movie **x** where none of the Bacon's acted. 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 ??, 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.

Exercise What does the datalog programs F2 compute?

```

F1(x) :- R1(x), not R2(x)
F2(x) :- R1(x), not F1(x)

```

Exercise The relation `Parent(x,y)` says that `x` is the parent of `y`. For example the query below finds Mary's children:

```

M(x) :- Parent('Mary',x)

```

Write datalog programs to compute the following: (1) All siblings of 'Alice'. (2) All cousins of 'Bob'. (3) All grandchildren of 'Carol'. (4) All Diane's children that have no children themselves. (5) All people that have no children without children.

3.5 Safe Datalog

What do these datalog queries compute?

```

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-recursive datalog with negation, to RA.

We start with a single rule:

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

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

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

For example, consider the rule:

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

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, \dots, 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, \dots, H_{i-1} , for which we simply substitute with their definitions.

Exercise: Translate Q4 and Q5 to RA. Translate the five queries you wrote earlier on the **Parent** table to RA.

4 The Relational Calculus

The Relational Calculus (RC) is the same as First Order Logic, and the same as Predicate Logic. We will cover RC only briefly in class, discussing 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. Actually, you already know RC! 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:

$$P ::= A \mid P \wedge P \mid P \vee P \mid \neg P \mid \exists x.P \mid \forall x.P \quad (2)$$

Here A stands for an atomic predicate, which is either a relational predicate or an interpreted predicate, as in ???. 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 ?? and \mathbf{x} are its free variables¹

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

Find all movies made in 1940:

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

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:

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

Exercise Write the SQL query from ?? in RC. Note: you need to use only existential quantifiers.

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:

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

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.

Given the **Parent**(\mathbf{x}, \mathbf{y}) relation, find all persons such that all their children have children²:

$$M(x) \equiv \forall y. \text{Parent}(x, y) \Rightarrow \exists z. \text{Parent}(y, z)$$

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

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

²This query is actually incorrect, because it is not domain independent (read on to find out what that means). Exercise: modify it to make it domain dependent.

Exercise Write the following queries in the relational 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 if 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:

$\text{Q5}(x) :- \text{Movie}(x, y, 1994), \text{Actor}(z, \text{'Kevin'}, \text{'Bacon'}), \text{not Casts}(z, x)$

This is equivalent to:

$\text{Q5}(x) \equiv \exists y. \exists z. (\text{Movie}(x, y, 1994) \wedge \text{Actor}(z, \text{'Kevin'}, \text{'Bacon'}) \wedge \neg \text{Casts}(z, x))$

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:

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

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 frequent 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:

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

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.

In Mathematics we don’t hesitate writing sentences that depend on the domain. For example we write $\forall x. \exists y. y < x$: is it 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.

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

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

Consider a Relational Calculus query, $??$. To translate it into non-recursive datalog with negation, start by writing down all its subexpressions, call them P_1, P_2, \dots, P_m . List smaller expressions first, then larger expressions. Make the following changes in the subexpressions P_1, \dots, P_m :

- Remove universal quantifiers by using de Morgan's laws: $\forall z.P_i \equiv \neg \exists z.\neg P_i$.
- Ensure that every negation occurs in a context $P_i \wedge \neg P_j$, and every free variable in P_j also occur in P_i . This is always possible³.

Let $\mathbf{x}_1, \dots, \mathbf{x}_m$ be the free variables in P_1, P_2, \dots, P_m . For each subexpression P_i , define a query H_i , for $i = 1, m$:

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

Next, translate every rule $H_i \equiv P_i$ into non-recursive datalog, depending on what operation P_i performs last (refer to $??$):

³For many queries in practice this assumption already holds. When it doesn't hold, we replace a subformula $\neg P_j$ with $D(z_1) \wedge \dots \wedge D(z_k) \wedge \neg P_j$, where z_1, \dots, z_k are the free variables in P_j and D is the formula computing the *active domain* of the database, that is, all the constants that occur in any attribute of any relation. For example, for the drinker-bars-beers example, the active domain is computed as follows:

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

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

$$H_i(\mathbf{x}_i) : \neg P(t_1, \dots, t_k)$$

Conjunction If $P_i = P_j \wedge P_k$ then the datalog rule is:

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

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 compute H_i using two datalog rules:

$$\begin{aligned} H_i(\mathbf{x}_i) &: \neg H_j(\mathbf{x}_j) \\ H_i(\mathbf{x}_i) &: \neg H_k(\mathbf{x}_k) \end{aligned}$$

Both rules are safe because $\mathbf{x}_i = \mathbf{x}_j = \mathbf{x}_k$. (Why ?)

Negation By our assumption negation occurs in the following context: $P_i = P_k \wedge \neg P_j$. The datalog rule is:

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

Existential quantifier If $P_i = \exists z. P_j$ then $\mathbf{x}_i = \mathbf{x}_j - \{z\}$ and the datalog rule is:

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

4.5 Example of a Translation

Consider the following query:

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

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

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

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

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

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

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

$$\begin{aligned} H_1(x, z) &\equiv \text{Frequents}(x, z) \\ H_2(y, z) &\equiv \text{Serves}(z, y) \\ H_3(x, y) &\equiv \text{Likes}(x, y) \\ H_4(x, y, z) &\equiv \text{Likes}(x, y) \wedge \text{Serves}(z, y) \wedge \neg \text{Frequents}(x, z) \\ H_5(x, y) &\equiv \exists z. (\text{Likes}(x, y) \wedge \text{Serves}(z, y) \wedge \neg \text{Frequents}(x, z)) \\ H_6(x, y) &\equiv \text{Likes}(x, y) \wedge \neg \exists z. (\text{Likes}(x, y) \wedge \text{Serves}(z, y) \wedge \neg \text{Frequents}(x, z)) \\ H_7(x) &\equiv \exists y. \text{Likes}(x, y) \wedge \neg \exists z. (\text{Likes}(x, y) \wedge \text{Serves}(z, y) \wedge \neg \text{Frequents}(x, z)) \end{aligned}$$

(We took one shortcut: the formula H_4 should be split into two: first $H'_4(x, y, z) = \text{Likes}(x, y) \wedge \text{Serves}(z, y)$, then $H_4(x, y, z) = H'_4(x, y, z) \wedge \neg \text{Frequents}(x, z)$. Instead, we lumped them together into a single formula, for simplicity.)

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

$$\begin{aligned} H_1(x, z) &: \neg \text{Frequents}(x, z) \\ H_2(y, z) &: \neg \text{Serves}(z, y) \\ H_3(x, y) &: \neg \text{Likes}(x, y) \\ H_4(x, y, z) &: \neg H_3(x, y), H_2(z, y), \text{ not } H_1(x, z) \\ H_5(x, y) &: \neg H_4(x, y, z) \\ H_6(x, y) &: \neg H_3(x, y), \text{ not } H_5(x, y) \\ H_7(x) &: \neg H_6(x, y) \end{aligned}$$

We can simplify this, by removing the IDB's H_4 and H_6 , writing:

$$\begin{aligned} H_1(x, z) &: \neg \text{Frequents}(x, z) \\ H_2(y, z) &: \neg \text{Serves}(z, y) \\ H_3(x, y) &: \neg \text{Likes}(x, y) \\ H_5(x, y) &: \neg H_3(x, y), H_2(z, y), \text{ not } H_1(x, z) \\ H_7(x) &: \neg H_3(x, y), \text{ not } H_5(x, y) \end{aligned}$$

Simplify it further, by removing H_1, H_2, H_3 :

$$\begin{aligned} H_5(x, y) &: \neg \text{Likes}(x, y), \text{Serves}(z, y), \text{ not } \text{Frequents}(x, z) \\ H_7(x) &: \neg \text{Likes}(x, y), \text{ not } H_5(x, y) \end{aligned}$$

The SQL expression follows easily from the datalog program:

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

Of course we don't need to eliminate duplicates before checking if a subquery is non-empty, so we can drop the **distinct** in subqueries. More subtly, **Likes l2** is actually not needed. The query can be further simplified, and becomes:

```
Q11: select distinct l.drinker
      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 $\exists z_1 \dots \exists z_m. P$, where P is the translation of R , and z_1, \dots, 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 $P_1 \wedge \neg P_2$.

Summarizing:

Theorem 4.1 *The following three languages can express exactly the same sets of queries: (1) Relational Algebra (RA), (2) non-recursive datalog with negation, (3) Relational Calculus (RC).*