

3

Queries

Structures and Relational Database Query Languages

Structures and relational databases

A relational language \mathbf{L} is a first order language with equality and with the following properties:

1. It has finitely many predicate symbols.
2. It has finitely many constant symbols c_1, \dots, c_n .

A relational database is a structure \mathbf{R} for \mathbf{L} such that:

1. \mathbf{R} has a finite domain C_1, \dots, C_n , with one domain element for each constant symbol of \mathbf{L} .
2. For each constant symbol c_i of \mathbf{L} , \mathbf{R} interprets c_i as C_i . That is, $c_i^{\mathbf{R}} = C_i$.

The education database is an example:

1. \mathbf{L} has the predicate symbols *Instructor*, *Enrolled*, *Prereq*, *Grade*, *PassingGrade*.
2. \mathbf{L} has the constant symbols $\{ray, hec, jill, \dots, cs200, cs230, \dots, a+, \dots, d\}$.
3. \mathbf{R} has the domain $\{Ray, Hec, Jill, \dots, CS200, CS230, \dots, A+, \dots, D\}$ and \mathbf{R} interprets *ray* as Ray, *hec* as Hec, *etc.*

Almost all database systems in use today (like SQL, Oracle, MS Access, Appleworks, *etc.*) are based on the theory of relational databases.

Queries and their answers

Let \mathbf{L} be a relational language.

Let x_1, \dots, x_n be distinct variables.

Let Q be a formula of \mathbf{L} whose free variables (if any) are among the x_i .

Then $\langle (x_1, \dots, x_n) : Q \rangle$ is a query of \mathbf{L} .

Let \mathbf{R} be a relational database for \mathbf{L} . The answer to the query $\langle (x_1, \dots, x_n) : Q \rangle$ with respect to \mathbf{R} is

$\{ (d_1, \dots, d_n) \mid Q \text{ is true in } \mathbf{R} \text{ for } (x_1, \dots, x_n) \text{ as } (d_1, \dots, d_n) \}$.

So the answer to a database query is a set of tuples taken from the domain of \mathbf{R} .

The set of all values of the variables x_1, \dots, x_n for which the formula Q is true.

An important special case: $n = 0$.

Then Q must be a sentence (no free variables).

In this case, there are two possible answers to Q :

$\{ () \}$, meaning *yes*, or

$\{ \}$, meaning *no*.

Queries for the educational database

Find all student-course pairs such that the student is enrolled in the course.

$\langle (s, c) : Enrolled(s, c) \rangle$

Answer: $\{ (Jill, CS230), (Jack, CS230), \dots, (Flo, M200) \}$

Find all students currently enrolled in some course.

$\langle (s) : \exists c Enrolled(s, c) \rangle$

Answer: $\{ (Jill), (Jack), \dots, (Flo) \}$.

Convention: When the query asks for all one-tuples, omit the parentheses.

$\langle s : \exists c Enrolled(s, c) \rangle$

Answer: $\{ Jill, Jack, \dots, Flo \}$.

Find all instructors teaching exactly one course.

$\langle i : \exists c [Instructor(i, c) \wedge \forall c' (Instructor(i, c') \supset c' = c)] \rangle$.

Answer: $\{ Ray, Hec, Pat \}$.

More example queries

Find all instructors teaching more than one course.

$$\langle i : \exists c [\text{Instructor}(i, c) \wedge \exists c' (\text{Instructor}(i, c') \wedge \neg c' = c)] \rangle.$$

Answer: {Sue}.

Which courses does Sue teach in which Ann is enrolled, but in which Jill is not enrolled?

$$\langle c : \text{Instructor}(\text{sue}, c) \wedge \text{Enrolled}(\text{ann}, c) \wedge \neg \text{Enrolled}(\text{jill}, c) \rangle.$$

Answer: {M100}.

Which courses does Sue teach in which Ann or Jill are enrolled?

$$\langle c : \text{Instructor}(\text{sue}, c) \wedge [\text{Enrolled}(\text{ann}, c) \vee \text{Enrolled}(\text{jill}, c)] \rangle.$$

Answer: {M100, M200}.

More example queries

Is Jill enrolled in CS230?

$$\langle () : \text{Enrolled}(\text{jill}, \text{cs230}) \rangle.$$

Answer: {()}, meaning *yes*.

Is Jill enrolled in a course that Tom is taking?

$$\langle () : \exists c (\text{Enrolled}(\text{jill}, c) \wedge \text{Enrolled}(\text{tom}, c)) \rangle.$$

Answer: {}, meaning *no*.

What are the non-passing grades?

$$\langle g : \neg \text{PassingGrade}(g) \rangle.$$

Answer: {Ray, Hec, Sue, Pat, CS230, CS238, ..., E, F}.

That is, D – the set of passing grades, where D is the domain of the structure.

This is a strange query since it actually asks for all domain elements of whatever sort (including people, courses) that are not passing grades.

Algorithms to compute answers

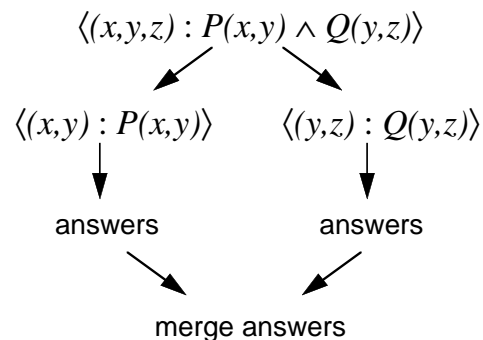
Given a query, $\langle (x_1, \dots, x_n) : Q \rangle$ and a database, we want to consider how to compute the answers (by hand or using a computer)

Brute force algorithm: Generate all n -tuples of domain elements, and keep those for which Q tests true.

Not computationally feasible for large databases (10^9 domain elements for genomic databases).

For $\langle (x_1, x_2, x_3) : Q \rangle$ this would require on the order of 10^{27} operations!

Something better: Recursively decompose complex queries into simple queries, answer these, and combine the resulting answers.



A recursive query evaluation algorithm

First, we consider complex queries:

1. $\text{ans}(\langle (x_1, \dots, x_n) : Q_1 \wedge Q_2 \rangle) = \text{ans}(\langle (x_1, \dots, x_n) : Q_1 \rangle) \cap \text{ans}(\langle (x_1, \dots, x_n) : Q_2 \rangle).$
2. $\text{ans}(\langle (x_1, \dots, x_n) : Q_1 \vee Q_2 \rangle) = \text{ans}(\langle (x_1, \dots, x_n) : Q_1 \rangle) \cup \text{ans}(\langle (x_1, \dots, x_n) : Q_2 \rangle).$
3. $\text{ans}(\langle (x_1, \dots, x_n) : \exists y Q \rangle) = \text{project}(\text{ans}(\langle (x_1, \dots, x_n, y) : Q \rangle)).$ assuming y is not among the x_i

Let S be a set of k -tuples for some fixed $k \geq 1$.

Then $\text{project}(S)$ is defined to be

$$\{(d_1, \dots, d_{k-1}) \mid \text{for some } d_k, (d_1, \dots, d_k) \in S\}.$$

In other words, to compute the projection of S , delete the last element of each k -tuple of S , and gather the resulting $(k-1)$ -tuples together.

4. $\text{ans}(\langle (x_1, \dots, x_n) : \neg Q \rangle) = D^n - \text{ans}(\langle (x_1, \dots, x_n) : Q \rangle).$

Here, D is the domain of the relational database, i.e., the set of all its domain elements.

This is a potential source of computational trouble: D^n can be huge!

Recursive query evaluation (cont.)

We have given recursive rules for complex queries with disjunction ($Q_1 \vee Q_2$), conjunction ($Q_1 \wedge Q_2$), negation ($\neg Q$), and existential quantification ($\exists y Q$).

1. To handle implication, ($Q_1 \supset Q_2$), replace this with the logically equivalent* ($\neg Q_1 \vee Q_2$).
2. To handle iff, ($Q_1 \equiv Q_2$), replace this with the logically equivalent $((Q_1 \wedge Q_2) \vee (\neg Q_1 \wedge \neg Q_2))$.
3. To handle universal quantification, $\forall x Q$, replace this with $\neg \exists x \neg Q$.

Question: Did we really need to give a decomposition rule for disjunction?

Now we know how to evaluate complex queries by recursively breaking them down to simpler queries, and combining the answers to these simpler queries.

When does this stop?

With atomic queries: atomic equality queries
or atomic relational queries

* Two sentences are logically equivalent iff they always have the same truth value. This will be defined precisely later.

Answering atomic equality queries

Atomic equality queries are queries of the form $\langle (x_1, \dots, x_n) : (t_1 = t_2) \rangle$ where each t is either one of the x 's or a constant symbol.

Examples:

$\langle x : m100 = x \rangle$

$\langle (x, y) : x = m100 \rangle$

$\langle (x, y) : x = y \rangle$

$\langle x : m100 = m200 \rangle$

$\langle () : m100 = m100 \rangle$

Answers:

$\{ M100 \}$

$\{ M100 \} \times D$

$\{ (d, d) \mid d \in D \}$

$\{ \}$

$\{ \langle () \rangle \}$

Atomic relational queries

Atomic relational queries are queries of the form $\langle (x_1, \dots, x_n) : A(t_1, \dots, t_k) \rangle$ where A is a k -ary predicate symbol, t_1, \dots, t_k are its arguments, and each t is either one of the x 's or a constant symbol.

Examples:

- $\langle (x, y) : \text{Enrolled}(x, y) \rangle$
- $\langle (x, y) : \text{Grade}(y, \text{cs230}, x) \rangle$
- $\langle (x, y) : \text{Enrolled}(x, \text{cs230}) \rangle$

An example where there are more variables in the tuple than in the formula component.

Such queries look strange, but they're legal, and we needed them in the recursive algorithm for

$$\langle (x, y) : \text{Enrolled}(x, \text{cs230}) \wedge \text{Grade}(x, \text{cs148}, y) \rangle$$

to get the students enrolled in CS230 with their grades from CS148.

- $\langle (x, y) : \text{SendsPackage}(x, y, x) \rangle$

$\text{SendsPackage}(x, y, z)$ might mean person x sends package y to person z .

The example illustrates how the same variable can occur more than once in a query body.

Answering atomic relational queries

By looking at the definition of truth, we can see that in general for an atomic relational query, we have that

$$\begin{aligned} \text{ans}(\langle (x_1, x_2, \dots, x_n) : A(t_1, \dots, t_k) \rangle) = \\ \{ (d_1, d_2, \dots, d_n) \in D^n \mid (|t_1|, \dots, |t_k|) \in A^R, \\ \text{where } |c| = c^R \text{ and } |x_i| = d_i \} \end{aligned}$$

Observe that if all the query variables x_i appear among the t 's, then the answer is no larger than A^R .

Such queries can be answered purely *locally*, by inspecting only the entries in the database table for A^R .

This would *not* be true for a query like

$$\langle (x, y) : \text{Prereq}(x, \text{cs230}) \rangle$$

which requires us to consider the entire domain.

However, as part of the query

$$\langle (x, y) : \text{Grade}(\text{sam}, x, y) \wedge \text{Prereq}(x, \text{cs230}) \rangle$$


we can see that once again the query could in principle be answered purely locally.

Problems with query evaluation

What are the sorts of queries that cause computational problems?

- wildcard variables:

$\langle (x, y) : \text{Prereq}(x, \text{cs230}) \rangle$

wildcard 

- some instances of the equality queries:

$\langle (x, y) : x = y \rangle$

- the negation rule

$$\text{ans}(\langle (x_1, \dots, x_n) : \neg Q \rangle) = D^n - \text{ans}(\langle (x_1, \dots, x_n) : Q \rangle)$$

These force us to inspect and retrieve from the entire database, which can be huge.

Question: Is there a class of queries that can be evaluated purely locally, and that includes most commonly occurring queries?

Answer: Yes, they are called *safe* queries.

Safe formulas

First, we define the formulas that are safe for a set of variables with the following rules:

1. Atomic formulas:
 - (a) When c and c' are constant symbols, $c = c'$ is safe for $\{ \}$.
 - (b) When c is a constant symbol and x a variable, $x = c$ and $c = x$ are safe for $\{x\}$.
 - (c) When x and y are variables, $x = x$ and $x = y$ are safe for $\{ \}$.
 - (d) When F is an atomic relational formula, and x_1, \dots, x_n are all the free variables of F , then F is safe for $\{x_1, \dots, x_n\}$.
2. If F is safe for V_F and G is safe for V_G , then $F \wedge G$ is safe for $V_F \cup V_G$.
3. If F is safe for V_F and G is safe for V_G , then $F \vee G$ is safe for $V_F \cup V_G$.
4. If F is safe for V_F and $x \in V_F$ then $\exists x F$ is safe for $V_F - \{x\}$.
5. If F is safe for V_F and G is safe for V_G , then $F \wedge \neg G$ is safe for $V_F \cup V_G$.
6. Nothing else is a safe formula.

Safe queries

$\langle (x_1, \dots, x_n) : Q \rangle$ is a safe query iff

1. Q is safe for $\{x_1, \dots, x_n\}$, and
2. x_1, \dots, x_n are free variables of Q .

So a safe query requires x_1, \dots, x_n to be *all and only* the free variables of Q .

Examples: all the queries given as examples for the educational DB on Slides 3–5, except for the last one.

Of these, only one might be troublesome:

Find all instructors teaching exactly one course.

$\langle i : \exists c [Instructor(i, c) \wedge \forall c' (Instructor(i, c') \supset c' = c)] \rangle$.

Rewrite this in a logically equivalent form to eliminate the implication and universal quantifier:

$\langle i : \exists c [Instructor(i, c) \wedge \neg \exists c' (Instructor(i, c') \wedge \neg c' = c)] \rangle$.

This query is safe (see the next slide).

But this is not safe (see the next slide):

$\langle (i, c') : \exists c [Instructor(i, c) \wedge \neg (Instructor(i, c') \wedge \neg c' = c)] \rangle$

Proofs of safety

Prove that the following query is safe:

$\langle i : \exists c [Instructor(i, c) \wedge \neg \exists c' (Instructor(i, c') \wedge \neg c' = c)] \rangle$

Proof: $c' = c$ is safe for $\{ \}$

$Instructor(i, c')$ is safe for $\{i, c'\}$

$Instructor(i, c') \wedge \neg c' = c$ is safe for $\{i, c'\}$

$\exists c' (Instructor(i, c') \wedge \neg c' = c)$ is safe for $\{i\}$

$Instructor(i, c)$ is safe for $\{i, c\}$

$Instructor(i, c) \wedge \neg \exists c' (Instructor(i, c') \wedge \neg c' = c)$
is safe for $\{i, c\}$

$\exists c [Instructor(i, c) \wedge \neg \exists c' (Instructor(i, c') \wedge \neg c' = c)]$
is safe for $\{i\}$

Since the query is over the variable i , and i is the only free variable, the query is safe.

Prove that the following query is not safe:

$\langle (i, c') : \exists c [Instructor(i, c) \wedge \neg (Instructor(i, c') \wedge \neg c' = c)] \rangle$

Proof: $Instructor(i, c') \wedge \neg c' = c$ is safe for $\{i, c'\}$ (as above)

$Instructor(i, c)$ is safe for $\{i, c\}$

$Instructor(i, c) \wedge \neg (Instructor(i, c') \wedge \neg c' = c)$
is safe for $\{i, c\}$

$\exists c [Instructor(i, c) \wedge \neg (Instructor(i, c') \wedge \neg c' = c)]$
is safe for $\{i\}$

Since the query is over the variables $\{i, c'\}$, the query is not safe.

Safe and unsafe queries

These queries are unsafe:

- $\langle x : x = x \rangle$.
- $\langle x : \exists y (x = y) \rangle$.
- $\langle (x,y) : x = y \rangle$.
- $\langle g : \neg \text{PassingGrade}(g) \rangle$.
- $\langle () : \exists g \neg \text{PassingGrade}(g) \rangle$.

But these are safe:

- $\langle g : \exists s \exists c [\text{Grade}(s,c,g) \wedge \neg \text{PassingGrade}(g)] \rangle$.
- $\langle (g,s) : \exists c [\text{Grade}(s,c,g) \wedge \neg \text{PassingGrade}(g)] \rangle$.
- $\langle c : \exists s \exists g [\text{Grade}(s,c,g) \wedge \neg \text{PassingGrade}(g)] \rangle$.

Safe queries are considered good because all the computations can be made *locally*.

The query can be answered by looking only inside the tables for the predicates mentioned in the query.

This can be proved, but we won't do that here. The proof is by *induction* using the rules defining safe formulas.

Note: There are queries that are not safe (as defined) but still might be answered locally:

$$\langle (x,y) : \text{Prereq}(x,cs230) \wedge x = y \rangle$$

Prolog

Prolog is a general purpose programming language that includes, among other things, facilities for computing safe queries for a relational database.

Prolog = PROgramming in LOGic.

The education database in Prolog:

```
instructor(ray,cs230).  instructor(sue,m100).
instructor(hec,cs230).  instructor(sue,m200).
instructor(pat,cs238).

enrolled(jill,cs230).  enrolled(jack,cs230).
    etc.
grade(sam,cs148,'a+').  grade(sam,cs148,d).
    etc.
```

Rules:

1. Relation names and domain element names must begin with a lower case letter.
2. Names that are not alphanumeric must be enclosed in single quotes.
3. Every entry must be terminated with a period.

Queries in Prolog

Prolog is an interactive programming language.

First, load your database as a file.

Next, interactively enter your queries.

Here is the Prolog version of $\langle (s,c) : \text{Enrolled}(s,c) \rangle$:

Variable names must begin with upper case. Predicate names must begin with lower case. A variable whose value will be the answer to the query. Must begin with upper case.

`setof([S,C], enrolled(S,C), Answer).`

Note the square brackets All inputs must terminate with a period.

The general pattern for a query in Prolog is

`setof(QueryVars, QueryBody, AnswerVariable).`

Find all one-tuples:

`setof(S, enrolled(S,m100), Ans).`

Yes/no queries:

`setof([], enrolled(tom,m100), A).`

The logical connectives in Prolog

Find all students currently enrolled in some course.

`setof(S, C^enrolled(S,C), Ans).`

$\exists c$

Which courses does Sue teach in which Ann is enrolled, but in which Jill is not enrolled?

`setof(C, (instructor(sue,C),
enrolled(ann,C),
not enrolled(jill,C)),
A).`

Note parentheses to delimit query body when it involves more than one element.

Translating to Prolog

Queries must be safe to guarantee that Prolog returns the correct answers.

Recall the query:

Find all instructors teaching exactly one course.

$\langle i : \exists c [Instructor(i, c) \wedge \forall c' (Instructor(i, c') \supset c' = c)] \rangle$.

We rewrote this in a logically equivalent form to eliminate the implication and universal quantifier:

$\langle i : \exists c [Instructor(i, c) \wedge \neg \exists c' (Instructor(i, c') \wedge \neg c' = c)] \rangle$.

This now easily translates into Prolog:

```
setof(I,
      C^(instructor(I,C),
          not C1^(instructor(I,C1),
                  not C1 = C)),
      Ans).
```

Executing queries in Prolog

What the user types in is in bold.

```
qew.cs> /local/src/cprolog/cprolog
C-Prolog version 1.4
```

First we load the file containing the relational database.

Here it is called "database".

```
| ?- [database].
database consulted 2500 bytes 0 sec.
yes
```

Next, we enter a query interactively.

```
| ?- setof(S,enrolled(S,m100),A).
S = _0
A = [ann,tom] ← The only variable that matters is the answer variable, here "A".
yes The "yes" means the query was answered successfully.
```

```
| ?- setof(S,C^enrolled(S,C),Ans).
S = _0
C = _1
Ans = [ann,bill,flo,jack,jill,may,sam,tom]
yes
```

Square brackets are Prolog's list notation.

Executing more Prolog queries

```
| ?- setof(C,(instructor(sue,C),
              enrolled(ann,C),
              not enrolled(jill,C)), A).
```

C = _0

A = [m100]

yes

```
| ?- setof([I,C],
          (instructor(I,C),
           not C1^(instructor(I,C1), not C1 = C)),
          A).
```

I = _0

C = _1

C1 = _11

A = [[hec,cs230],[pat,cs238],[ray,cs230]]

yes

A yes/no query

```
| ?- setof([], not enrolled(tom,m200), A).
```

A = [[]]

The answer is “yes”.

yes

```
| ?- setof([], enrolled(tom,m200), A).
```

no

*Prolog says “no” instead
of returning the empty list.*

Defining new relations

The predicate symbols for a relational database (*Grade*, *Prereq*, etc.) are called base predicates.

Often, one would like to define new concepts in terms of these old ones:

- A first class grade is A+, A, or A-.

$$\forall g[FirstClassGrade(g) \equiv \\ g = a+ \vee g = a \vee g = a-]$$

- A good student is one who has taken at least one course, and who has a first class grade in all the courses she has taken.

$$\forall s[GoodStudent(s) \equiv \exists c, g Grade(s, c, g) \wedge \\ \forall c, g (Grade(s, c, g) \supset FirstClassGrade(g))]$$

- A current student is one who is currently enrolled in a course.

$$\forall s[CurrentStudent(s) \equiv \exists c Enrolled(s, c)]$$

- Good current students:

$$\forall s[GoodCurrentStudent(s) \equiv \\ CurrentStudent(s) \wedge GoodStudent(s)]$$

Defining new relations in Prolog

Predicate names must begin with lower case.
 Universally quantified variables begin with upper case.

\equiv
`goodCurrentStudent(S) :-`
 \wedge
`currentStudent(S), goodStudent(S).`
 The body of a safe query follows the :-
 The body terminates with a period.

\vee
`firstClassGrade(G) :-`
`G = 'a+' ; G = 'a' ; G = 'a-'.`

`currentStudent(S) :- C^enrolled(S,C).`
`goodStudent(S) :- C^G^grade(S,C,G),`
`not C1^G1^(grade(S,C1,G1),`
`not firstClassGrade(G1)).`

Note: Prolog wants all existentially quantified variables within a query or definition to have unique names. So this will not work correctly:

```
goodStudent(S) :- C^G^grade(S,C,G) ,
not C^G^(grade(S,C,G), not firstClassGrade(G)).
```

Executing queries with defined relations

```
gew.cs> /local/src/cprolog/cprolog
C-Prolog version 1.4
```

First we load the files. Here the relational database is in one file and the definitions are in another, but they can all be in one.

```
| ?- [database, definitions].
database consulted 1940 bytes 1.862e-09 sec.
definitions consulted 560 bytes 1.862e-09 sec.
yes
```

```
| ?- setof(S,goodCurrentStudent(S),Answers).
S = _0
Answers = [may,sam]
yes
```

```
| ?- setof([S,I],
  (goodCurrentStudent(S),
   C^(enrolled(S,C), instructor(I,C))),
  A).
S = _0
I = _1
C = _10
A = [[may,pat],[sam,hec],[sam,ray],[sam,sue]]
yes
```