Writing Queries

Christoph Koch

How to write a query

- Read the problem statement attentively.
- Parse it.
- Note subqueries, existential/universal quantifiers.
  - Universal: „all“, „every“, „only“, ... But not „Find all tuples such that ...“.
- If there are universal quantifiers, you may find it helpful to first write a calculus or SQL query, even if the goal is an algebra query.
  - First write calculus and then map it to algebra.

Calculus Queries

- You may know this as „predicate logic“ or „first-order logic“.

- Remember safety/range-restriction:
- „Forall x: R(x)“ does not make sense.

From English to “Quantified English”

1. Are all students CS students?
2. Is it true that, for all things x, if (x is a student) then (x is a CS student) ?

$$\forall x(Student(x) \Rightarrow CSStudent(x))$$

-- The parentheses are here to help you parse the sentence.

From English to “Quantified English”

1. Return all students who have taken only CS courses.
2. Return all students who have not taken a course that is not a CS course.
3. Return all x such that ((x is a student) and (there does not exist a y such that (x has taken y) and (y is not a CS course))).

From English to “Quantified English”

- Return the oldest student(s).
- Schema: Student(sid, age)

1. Return a student if there is no older student.
2. Return an x if (x is a student) and (there does not exist a y such that ((y is a student) and (y is older than x)).
   -- Unfortunately our schema is not Student(sid), Older(sid1, sid2).
3. Return an x if there is a v such that (x is a student aged v) and (there do not exist y, w such that ((y is a student aged w) and (w is greater than v)).
4. {x | exists v: Student(x,v) and not exists y,w: ((Student(y,w) and w > v))}.

From English to “Quantified English”

1. Return all the students who have taken all the required courses.
2. Return the students who have taken all the required courses.
3. Return x if ((x is a student) and (there does not exist a y such that (y is a required course) and (x has not taken y))).
4. {x | Student(x) and not exists y (Req(y) and not Taken(x,y))}

From English to “Quantified English”

1. Return all pairs of students who have taken the same courses.
2. Return the pairs of students (x, y) such that, for all courses z, whenever x has taken z then y has also taken z and whenever y has taken z then x has also taken z.
3. Return all pairs (x,y) such that ((x is a student) and (y is a student) and, for all z, (((x has taken z) implies (y has taken z)) and ((y has taken z) implies (y has taken z))))

From English to “Quantified English”

1. (Check the following:) Only you can grasp the calculus.
2. For all x, if x can grasp the calculus, then x is you.
3. Forall x: (CanGraspCalc(x) => x=you)

or

- There does not exist anyone who can grasp the calculus [and] who is different from you.
- Not exists x: (CanGraspCalc(x) and not x=you)

From English to Calculus to SQL

- Output all sailors who have <u>only</u> sailed in red boats.
- Schema S(S), R(S,B), B(B, C)

1. Output all sailors who have not sailed in a boat that is not red.
2. Output all sailors for whom there does not exist a boat that they have sailed and that is not red.
3. { s | S(s) and not exists b,c: B(b, c) and R(s, b) and c != „red“}
4. { ss | S(ss) and not exists bb,bc,rs,rb: B(bb, bc) and R(rs, rb) and rs=ss and bb=rb and bc != „red“ }
5. SQL:

```
select S.S from S
where not exists
  (select * from B, R
   where R.S  = S.S
   and   B.B  = R.B
   and   B.C != „red“);    -- note the similarity to 4 !
```

From English to Calculus to SQL

1. Return the students who have taken all the required courses. (This is an example from above.)
2. {s | Student(s) and not exists c (Req(c) and not Taken(s,c))}
3. {ss | Student(ss) and not exists rc, ts, tc: (Req(rc) and not (Taken(ts,tc) and ss=ts and rc=tc))}
4.
```
select S.S from Student S
where not exists
 (select * from Req R
  where not exists
   (select * from Taken T
    where S.S=T.S and R.C=T.C));
```

Calculus reformulations

- Some rules (there are of course many more):
  forall x: phi(x) = not exists x: not phi(x).
  not forall x: not psi(x) = exists x: psi(x).
  not(A and B) = (not A) or (not B) (DeMorgan‘s law)
  A => B = (not A) or B
  (Not A) or (Not B) or C = not (A and B) or C = (A and B) => C

- Output all sailors who have <u>only</u> sailed in red boats.

1. { s | S(s) and not exists b,c: B(b, c) and R(s, b) and c != „red“}
2. { s | S(s) and forall b,c: not (B(b, c) and R(s,b) and c != „red“)}
3. { s | S(s) and forall b,c: (not B(b, c)) or (not R(s,b)) or c = „red“ }
4. { s | S(s) and forall b,c: (B(b, c) and R(s,b)) => c = „red“}

- Output all sailors for whom it is true that all the boats they have sailed are red.

Natural Language Ambiguity

- (Is it true that) everybody loves somebody sometimes?
- Schema:
  Person(person), R(lovingperson, lovedperson, timestamp)

1. forall x: (Person(x) => exists y exists z: R(x,y,z))
   or
2. exists y forall x (Person(x) => exists z: R(x,y,z))
   or
3. exists y exists z forall x: (Person(x) => R(x,y,z))

- In (1), two x can have different love interests, and each x can love different people at different times.
- In (2) there is a single y such that, at possibly different times, everybody loves that y.
- In (3) there is a single y and a particular time z such that everyone loves y at that time point z.

English to Calculus to SQL

- (Is it true that) everybody loves somebody sometimes?
- Schema:
  Person(person), R(lovingperson, lovedperson, timestamp)

1. forall pp: (Person(pp) => exists rp1, rp2, rt: (R(rp1,rp2,rt) and pp=rp1))
2. not exists pp: (Person(pp) and not exists rp1, rp2, rt: (R(rp1,rp2,rt) and pp=rp1))

- ```
  select ‘yes’ from Dummy where not exists
    (select * from Person where not exists
      (select * from R
       where Person.person=R.lovingperson));
  ```
- Dummy can be any nonempty relation.
  ```

Quantifiers on empty sets

- Forall x: R(x) => phi(x)
  - If R is empty, then this is true.
  - All CS students love CS432: If there are no CS students, then this is „vacuously“ true.
- „Forall“ is like „and“, „Exists“ is like „or“.
- Phi_1 and ... and Phi_n: if n=0 then this is true.
- Phi_1 or ... or Phi_n: if n=0 then this is false.
- x_1 * ... * x_n: if n=0 then this is 1
- x_1 + ... + x_n: if n=0 then this is 0

Summary: Expressive Power

- Calculus and Algebra equivalent
- Aggregates only in SQL.
- But Min, Max queries can be rewritten so as not to use aggregate operators.