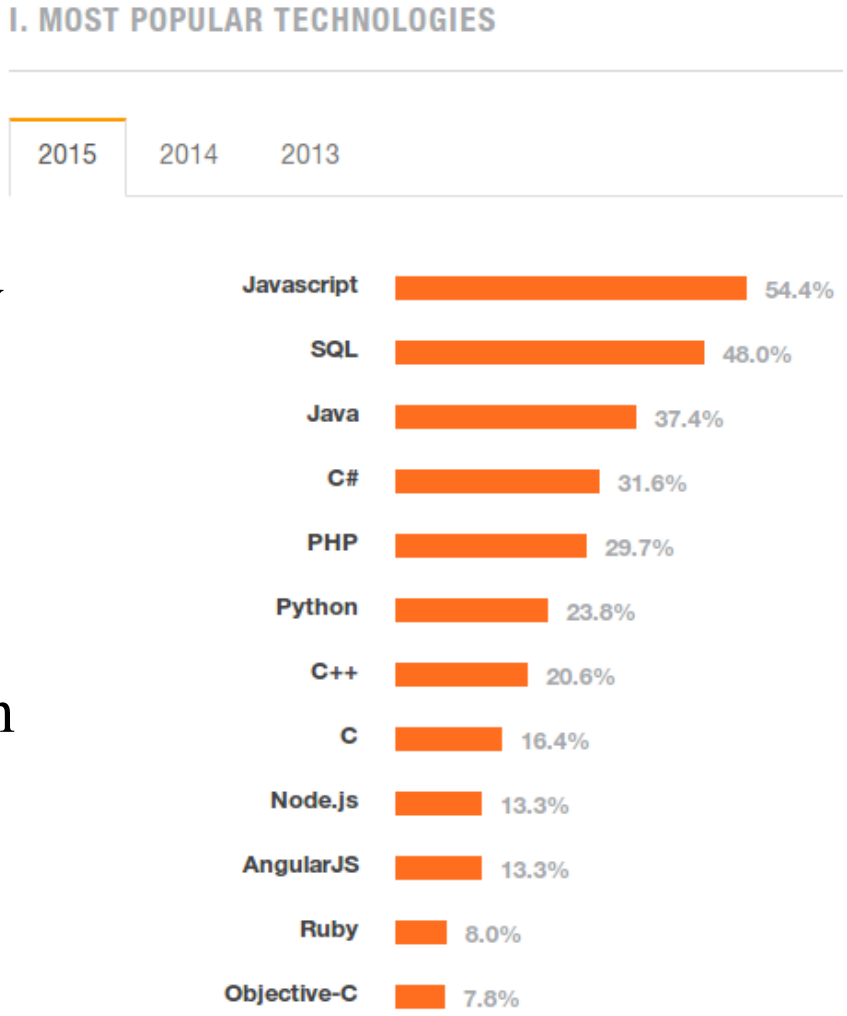
Writing Queries

Christoph Koch

Why do you need to be able to write database queries?

- So you like machine learning… ☺
- However, most data analysis today is in the form of (SQL) queries.
- Out in industry, you'll need to be able to write SQL queries confidently.
- You can do surprisingly much with queries.



http://stackoverflow.com/research/developer-survey-2015

Things you can do in SQL

- All of first-order predicate logic, and more.
- Linear algebra: matrix multiplication, ⋯
- Statistics, e.g. moments.
- Data integration (schema mapping).

- SQL is a domain-specific language (DSL)
  - Limited (non-Turing complete lang) specialized for writing queries.
  - Allows to write lots of typical queries with very little code.
- Declarative: don’t worry about how to execute it!

SQL Example: Matrix multiplication

Two matrices (schemata):
A(i,j,aij), B(j,k,bjk)

```
select    i, k, sum(aij * bjk)
from      A, B
where     A.j = B. j
group by A.j
```

Plan for this lecture

- How to write a complex query

- Queries that are expressible in first-order logic
- Inexpressible queries; aggregate queries

- We don’t cover or assume exotic SQL features (you read the manual for that).
- But: We delve deeply into the fundamental power of *declarative* languages such as SQL, and their *compositionality*.

How to write a query

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

Relational Calculus Queries

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

- Safety/range-restriction:
  - „`forall x: R(x)`“ does not make sense.
  - „`forall x: P(x) => R(x)`“ is ok.

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 in (2) are here to help you parse the sentence.
- Introduce variables to avoid ambiguity.

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))).

- Rewrite rule for “only” => there does not exist a counterexample.

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 (x has taken z))))

From English to “Quantified English”

1. (Is it true that:) 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 only 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 only 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 distinct '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

Building complex queries from smaller parts

- Express the query as a sequence of steps/ views.
- Define the views.
- SQL, calculus, relational algebra are compositional: If the solution is not to use views, compose them into a single query.

A complicated query

- Schema:
  - “student” S(sid),
  - “course” C(cid),
  - “course taken” T(sid, cid)
- Find the students who take at least one course for which the size of the group of courses that are taken by the same group of students who also take this course is maximal.

A complicated query #2

- Find the students who take at least one course x for which the size of the group of (courses that are taken by the same group of students) who also take x is maximal.
- Ceq := (pairs of) courses that are taken by the same group of students.

```
{ (x,y) | forall s: Course(x) and Course(y) and
          (Taken(s, x) <=> Taken(s, y)) }

{ (x,y) | Course(x) and Course(y)
    and not exists s: not
          (((not Taken(s, x)) or Taken(s, y))  and
          (Taken(s, x) or (not Taken(s, y)))) }
```

A complicated query #3

Domain relational calculus:

```
Ceq := { (x,y) | Course(x) and Course(y)
    and (not exists s:
        (Taken(s, x) and (not Taken(s, y))))
    and (not exists s:
        (Taken(s, y) and (not Taken(s, x)))) }
```

A complicated query #4

Tuple relational calculus:

```
Ceq := { (c1.cid, c2.cid) | (c1 in Course) and (c2 in Course)
    and not exists (
       (t1 in Taken) and (t1.cid = c1.cid) and
       (not exists ((t2 in Taken) and
                    (t2.cid = c2.cid) and (t1.sid = t2.sid))))
    and not exists (
       (t2 in Taken) and (t2.cid = c2.cid) and
       (not exists ((t1 in Taken) and
                    (t1.cid = c1.cid) and (t1.sid = t2.sid))))
}
```

A complicated query #5

```
create view ceq as (
select c1.cid, c2.cid from Course c1, Course c2
where not exists (
   select * from Taken t1
   where t1.cid = c1.cid and not exists (
      select * from Taken t2
      where t2.cid = c2.cid and t1.sid = t2.sid))
and not exists (
   select * from Taken t2
   where t2.cid = c2.cid and not exists (
      select * from Taken t1
      where t1.cid = c1.cid and t1.sid = t2.sid))
)
```

A complicated query #6

- Find the students who take at least one course x for which the (size of the equivalence class of x with respect to Cequiv) who also take x is maximal.
- Remember: Cequiv is an equivalence relation – it is reflexive, symmetric, and transitive

- Assume the schema of Cequiv is (c1, c2)
- size of the equivalence class of x with respect to Cequiv:

```
create view grpsize as (
select c1, count(c2) from Cequiv group by c1
)
```

A complicated query #7

- Find the students who take at least one course x for which grpsize(x, s) is such that s is the maximal groupsize).
- Schema grpsize(cid, size)
- Cwmaxgrpsize: courses with maximal group size.

```
create view Cwmaxgrpsize as
(select cid from grpsize where
size = (select max(size) from grpsize)
)
```

A complicated query #8

- Find the students who take at least one course x in cwmaxgrpsize.

```
select S.sid
from   Student S, Taken T,
       Cvmaxgrpsize C
where  S.sid = T.sid
and    T.cid = C.cid
```

Expressive Power of Queries

- Calculus and Algebra equivalent
- Aggregates only in SQL.
- Counting can express universal quantification, but this is brittle (set vs. Bag semantics)
  - Aggregates can do additional things not expressible in the calculus.
- But Min, Max queries can be rewritten so as not to use aggregate operators.

Universal quantification by counting

Students who have taken all courses:

```
select S.sid
from Student S
where (select count(*) from Taken T
       where T.sid = S.sid) =
      (select count(*) from Course
```

So, what are the pitfalls here?

Inexpressibility

- SQL is NOT a general-purpose (Turing-complete) programming language.
- Every SQL query can be implemented in C, Java, Python, ⋯ using a FIXED number k of nested loops over the database:

```
foreach t1 in DB-Tuples do {
    … {
            foreach tk in DB-Tuples do {
                some code without loops, recursion
        }
} … }
```

- There are even many polynomial-time problems that cannot be expressed.

Inexpressibility Example

- Example: transitive closure in a graph given by the edge relation.
- Need to be careful: Sometimes a problem may seem to need a powerful language feature like transitive closure, but there is a different way to express the question that does not need it.
  - Example: Cyclicity of a graph given by the reachability relation between nodes.

Expressible?

- Schema Pearl(P), Linked(P1, P2).

- Are all the pearls part of closed chains (graph-theoretically, cycles)?
  - Expressible in SQL
- Do the pearls form a single chain, i.e., are they all connected, but not as a cycle?
  - Not expressible in SQL

Expressible?

- Factorial of n?
  - No, but if you are given 1,…,n in a relation, it's expressible with agg-sum, log and exp (or a custom agg-multiplication).
- Graph 3-colorability (NP-complete)?
  - SQL expresses only a (small) fragment of PTIME. So: obviously no (assuming P <> NP).
  - If you encode the problem in the query rather than the database, you can express much more (PSPACE-complete problems) => query complexity.

Classroom task

- Find the students who only take courses that are taken by all students.

- Schema:
  - “student” S(sid),
  - “course” C(cid),
  - “course taken” T(sid, cid)