Data Program Synthesis: Work for week of 5/26

Updated May 27th, 2014

CURRENT GOAL FOR WEEK OF 5/26: AUTOMATICALLY FORMULATING SQL QUERIES

Consider a SQL query on two tables: *Sailors* and *ReservedBoats*.

Sailors

name	**age**	**rating**
Rusty	27	9
Billy	24	6
Old Whiskers	52	10

ReservedBoats

name	**color**	**sname**
SS Minnow	Red	Rusty
SS Barnacle	Blue	Billy
SS Rum	Red	Billy

The user wants to produce a certain output table:

Desired Output

RB.name	**RB.color**	**S.sname**
SS Barnacle	Blue	Billy

One correct way to produce this result is with the following query:

SELECT RB.name, RB.color, S.sname FROM Sailors S, ReservedBoats RB WHERE S.name = RB.sname AND S.name = "Billy"

Another reasonable way is using this query:

SELECT RB.name, RB.color, S.sname FROM Sailors S, ReservedBoats RB WHERE S.name = RB.sname AND RB.color = "Blue"

Unfortunately, the user is extremely lazy. He just wants to write this query:

SELECT ?+ FROM Sailors S, ReservedBoats RB WHERE ?+

In other words, the user is going to indicate the tables involved, and indicates that the SELECT clause should have at least one element, and that the WHERE clause should have at least one element. The user also indicates the desired output table.

Our system' goal is to finish the user's query. It should fill in the blanks on the user's query to produce the desired output table.

Here's how we do it.

Step 0. INPUT

Input to the solver consists of:

- A schema of several tables. A schema is a set of tables. Each table has a set of (attr, type) pairs.
- A "wildcarded query". For now you can assume this query is fixed. It should take the form **SELECT ?+ FROM *AllRelations* WHERE ?+**
- Actual data for each table in the input schema
- Schema for the desired output table
- Data for the desired output table

STEP 1. UNFOLD QUERY INTO OPTIONS
The user's query: SELECT **?+** FROM Sailors, Reserved Boats WHERE **?+**

describes a huge number of possible queries. For example, the above query unfolds into:

SELECT (S.Name?, S.Age?, S.Rating?, RB.name?, RB.color?, RB.reservedSailor?) FROM Sailors S, ReservedBoats RB WHERE XXX

How many possible elements in the SELECT clause? Simply (NumAttrs)

XXX = {AllUnaryTestsInCartesianProduct, AllBinaryTestsInCartesianProduct}
AllUnaryTests = {S.Name = CONST, S.Name > CONST, S.Name, < CONST,

S.Age = CONST, S.Age > CONST, S.Age < CONST,
S.Rating = CONST, S.Rating > CONST, S.Rating < CONST,
RB.Name = CONST, RB.Name > CONST, RB.Name < CONST,
RB.color = CONST, RB.color > CONST, RB.color < CONST,
RB.sname = CONST, RB.sname > CONST, RB.sname < CONST}

*How many possible unary tests in the WHERE clause? (NumAttrs * NumOps * NumConsts)*

AllBinaryTestsInCartesianProduct = {S.Name {=,>,<} [S.name, S.Age, S.Rating, RB.name, RB.color, RB.sname],
S.Age {=,>,<} [S.name, S.Age, S.Rating, RB.name, RB.color, RB.sname],
S.Rating {=, >, <} [S.name, S.Age, S.Rating, RB.name, RB.color, RB.sname],
RB.name, {=, >, <} [S.name, S.Age, S.Rating, RB.name, RB.color, RB.sname],
RB.color, {=, >, <} [S.name, S.Age, S.Rating, RB.name, RB.color, RB.sname],
RB.sname, {=, >, <} [S.name, S.Age, S.Rating, RB.name, RB.color, RB.sname]}

*How many possible binary tests in the WHERE clause? (NumAttrs * NumOps * NumAttrs)*

The number of possible clauses is the **sum** of the above quantities. (NumAttrs + (NumAttrs * NumOps * NumConsts) + (NumAttrs * NumOps * NumAttrs).

The number of possible queries is the **product** of the above quantities. (NumAttrs * (NumAttrs * NumOps * NumConsts) * (NumAttrs * NumOps * NumAttrs)) == (NumAttrs^4 * NumOps^2 * NumConsts)

The NumAttrs is fixed by the input schema. The NumOps is fixed by the query language. The NumConsts is described below. For realistic but still simple tables, NumAttrs might be as much as 100. Figure NumOps will be about 5. That's (100^4 * 5^2 * NumConsts) == 2.5B * NumConsts possible programs.

Our goal is to pick the best query or queries from this total set of possibilities.

Step 2. GENERATE CONST CANDIDATES

What should the CONST values be? Include:

- 0
- 1
- -1
- 2
- MAX, MIN, AVG, SUM of every column
- COUNT of table
- Every constant value in the desired output

Use these to "unfold" the unary tests into a larger set.

Step 3. DISQUALIFY TYPE VIOLATIONS

A bunch of these tests are impossible, because they violate type constraints. E.g., you can't test equality between a string and an integer. Throw out those unary tests and binary tests that violate type constraints.

Step 4. GENERATE A SYSTEM OF LOGICAL FORMULAE

Imagine that for every candidate clause we create a boolean variable. If this variable is true, then the candidate clause is in the answer query. If the variable is false, then the clause is not present. There should be up to (NumAttrs + (NumAttrs * NumOps * NumConsts) + (NumAttrs * NumOps * NumAttrs) of these variables (not including the ones we threw out in Step 2). Let's call the total number of clauses C. That means there's x_0, x_1, … x_{|C|-1} boolean vars.

We eventually want to come up with a "correct query" by figuring out which of these variables should be true. If the var is true, then the clause is in the answer. What are the logical constraints on this set of variables?

1. Well, one is that at least one of the SELECT clause variables is true. So create SELECT VARIABLES sv_0, sv_1, … sv_|S|-1 to indicate whether the corresponding SELECT CLAUSE is present in the query. Then set up some logical constraints. We want at least one such clause in the output, so the formula we want to express is: (sv_0 v sv_1 v …. sv_{|S|-1}).
2. Another is that at least one of the WHERE clause variables is true. So for WHERE VARIABLES, set up a formula similar to the one in Step #1.
3. We also want some variables that indicate which to make sure the output looks like we want it to look. So for each column in cartesian product of input tables, create an OUTPUT SCHEMA VARIABLE that indicates whether that attribute is in the output. Call them variables os_0, os_1, … os_{|S|-1}. We also want to logically hook up the output schema variable to the query. (sv_0 => os_0) ^ (sv_1 => os_1), …. Saying (sv_0 => os_0) <==> (~sv_0 v os_0)
4. Set up an OUTPUT TUPLE VARIABLE for each record in the cartesian product of the input tables. Call the variables otv_0, otv_1, … otv_{|OTV|-1}. If the variable is true, it indicates that the corresponding tuple is in the output. There will be a LOT of these. Also set up a formula that logically hooks up these output variables to the query. So if WHERE CLAUSE VARIABLE wv_i will emit the cartesian product tuple j, create a formula (wv_i => ot_j) <==> (~wv_i v ot_j). You will have to scan the input data and apply all possible where clauses in order to figure out this set of clauses.
5. We also want to say that if a tuple is present in the output, then SOME query clause produced it. So if OUTPUT TUPLE j can be produced by WHERE CLAUSE VARIABLES a, b, or c, then we write: (otv_j => (wv_a v wv_b v wv_c)) <==> (~otv_j v wv_a v wv_b v wv_c).
6. Now create DESIRED OUTPUT CLAUSES. These clauses are true because they match what's in the desired output table. For each attribute a in the output table that we want, write down (sv_a). For each tuple T in the desired output, enumerate all the tuples in the cartesian product that contain all the values in T. If T is produced by output tuples a, b, and c, we can write: (outv_a v otv_b v otv_c). You will have to do scan all the data in order to produce this set of clauses.

will have to do scan all the data in order to produce this set of clauses.

7. Finally, we want to say that output attributes and output tuples are not desired if they are not in the output set. So for each attribute a in the cartesian product that is NOT in the output table, write down (~ sv_a). For each tuple x in the cartesian product that DOES NOT contain all the values in any tuple in T, write down (~ otv_x).

This step actually produces two artifacts:

1. The set of formulas
2. The mapping between the CANDIDATE SQL PIECEs and the SELECT VARIABLES and the WHERE VARIABLES.

We'll find a solution to item 1. Then we'll need data item 2 in order to decode the solution and produce a user-comprehensible query.

Step 5. TRANSLATE THE FORMULAE INTO CNF

A CNF formula is a logical formula over boolean variables that takes the following form:
(x1 v x2 v x3) ^ (x1 v x4 v x5) ^ …..

All of the above clauses are disjunctions. So creating a CNF just means ANDing them together.

Step 6. SEND THE CNF PROBLEM TO A SAT SOLVER

A SAT Solver is a system that takes a problem in CNF and formulates a truth assignment to the variables such that the formula is true. It just solves the CNF.

Step 7. OBTAIN SOLUTION(S) AND SHOW TO USER

Remember that each variable in the CNF corresponds to a CANDIDATE SQL PIECE. Take the GRAPH VARIABLES that are true and look up the corresponding CANDIDATE SQL PIECE in the mapping from Step 4. Show the resulting CANDIDATE SQL PIECES to the user: that's the answer!