Deductive Databases

Chapter 25

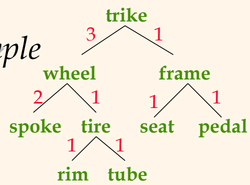
Motivation

- SQL-92 cannot express some queries:
  - Are we running low on any parts needed to build a ZX600 sports car?
  - What is the total component and assembly cost to build a ZX600 at today's part prices?
- Can we extend the query language to cover such queries?
  - Yes, by adding recursion.

Datalog

- SQL queries can be read as follows:
  "If some tuples exist in the From tables that satisfy the Where conditions,
  then the Select tuple is in the answer."
- Datalog is a query language that has the same if-then flavor:
  - New: The answer table can appear in the From clause, i.e., be defined recursively.
  - Prolog style syntax is commonly used.

Example



- Find the components of a trike?
- We can write a relational algebra query to compute the answer on ***the given instance of Assembly***.
- But there is no R.A. (or SQL-92) query that computes the answer on ***all Assembly instances***.

part	subpart	number
trike	wheel	3
trike	frame	1
frame	seat	1
frame	pedal	1
wheel	spoke	2
wheel	tire	1
tire	rim	1
tire	tube	1

Assembly instance

The Problem with R.A. and SQL-92

- Intuitively, we must join Assembly with itself to deduce that trike contains spoke and tire.
  - Takes us one level down Assembly hierarchy.
  - To find components that are one level deeper (e.g., rim), need another join.
  - To find all components, need as many joins as there are levels in the given instance!
- For any relational algebra expression, we can create an Assembly instance for which some answers are not computed by including more levels than the number of joins in the expression!

A Datalog Query that Does the Job

```
Comp(Part, Subpt) :- Assembly(Part, Subpt, Qty).
Comp(Part, Subpt) :- Assembly(Part, Part2, Qty),
                     Comp(Part2, Subpt).
```

head of rule **implication** **body of rule**

Can read the second rule as follows:
"**For all** values of Part, Subpt and Qty,
if there is a tuple (Part, Part2, Qty) in Assembly
and a tuple (Part2, Subpt) in Comp,
then there must be a tuple (Part, Subpt) in Comp."

Using a Rule to Deduce New Tuples

- Each rule is a ***template***: by assigning constants to the variables in such a way that each body "literal" is a tuple in the corresponding relation, we identify a tuple that must be in the head relation.
  - By setting Part=trike, Subpt=wheel, Qty=3 in the first rule, we can deduce that the tuple <trike,wheel> is in the relation Comp.
  - This is called an **inference** using the rule.
  - Given a set of tuples, we **apply** the rule by making all possible inferences with these tuples in the body.

Example

- For any instance of Assembly, we can compute all Comp tuples by repeatedly applying the two rules. (Actually, we can apply Rule 1 just once, then apply Rule 2 repeatedly.)

trike	spoke
trike	tire
trike	seat
trike	pedal
wheel	rim
wheel	tube

Comp tuples got by applying Rule 2 once

trike	spoke
trike	tire
trike	seat
trike	pedal
wheel	rim
wheel	tube
trike	rim
trike	tube

Comp tuples got by applying Rule 2 twice

Datalog vs. SQL Notation

- Don't let the rule syntax of Datalog fool you: a collection of Datalog rules can be rewritten in SQL syntax, if recursion is allowed.

```
WITH RECURSIVE Comp(Part, Subpt) AS

(SELECT A1.Part, A1.Subpt FROM Assembly A1)
UNION
(SELECT A2.Part, C1.Subpt
 FROM Assembly A2, Comp C1
 WHERE A2.Subpt=C1.Part)

SELECT * FROM Comp C2
```

Fixpoints

- Let f be a function that takes values from domain D and returns values from D. A value v in D is a fixpoint of f if f(v)=v.
- Consider the fn *double*+, which is applied to a set of integers and returns a set of integers (I.e., D is the set of all sets of integers).
  - E.g., *double+({1,2,5})={2,4,10} Union {1,2,5}*
  - The set of all integers is a fixpoint of *double*+.
  - The set of all even integers is another fixpoint of *double*+; it is smaller than the first fixpoint.

Least Fixpoint Semantics for Datalog

- The least fixpoint of a function f is a fixpoint v of f such that every other fixpoint of f is smaller than or equal to v.
- In general, there may be no least fixpoint (we could have two minimal fixpoints, neither of which is smaller than the other).
- If we think of a Datalog program as a function that is applied to a set of tuples and returns another set of tuples, this function (fortunately!) always has a least fixpoint.

Negation

Big(Part) :- Assembly(Part, Subpt, Qty), Qty >2, not Small(Part).
Small(Part) :- Assembly(Part, Subpt, Qty), not Big(Part).

- If rules contain **not** there may not be a least fixpoint. Consider the Assembly instance; trike is the only part that has 3 or more copies of some subpart. Intuitively, it should be in Big, and it will be if we apply Rule 1 first.
  - But we have Small(trike) if Rule 2 is applied first!
  - There are two minimal fixpoints for this program: Big is empty in one, and contains trike in the other (and all other parts are in Small in both fixpoints).
- Need a way to choose the intended fixpoint.

Stratification

- T <u>depends on</u> S if some rule with T in the head contains S or (recursively) some predicate that depends on S, in the body.
- <u>Stratified program:</u> If T depends on **not** S, then S cannot depend on T (or **not** T).
- If a program is stratified, the tables in the program can be partitioned into strata:
  - Stratum 0: All database tables.
  - Stratum I: Tables defined in terms of tables in Stratum I and lower strata.
  - If T depends on **not** S, S is in lower stratum than T.

Fixpoint Semantics for Stratified Pgms

- The semantics of a stratified program is given by one of the minimal fixpoints, which is identified by the following operational defn:
  - First, compute the least fixpoint of all tables in Stratum 1. (Stratum 0 tables are fixed.)
  - Then, compute the least fixpoint of tables in Stratum 2; then the lfp of tables in Stratum 3, and so on, stratum-by-stratum.
- Note that Big/Small program is not stratified.

Aggregate Operators

```
SELECT A.Part, SUM(A.Qty)
FROM Assembly A
GROUP BY A.Part
```

```
NumParts(Part, SUM(<Qty>)) :- Assembly(Part, Subpt, Qty).
```

- The **< … >** notation in the head indicates grouping; the remaining arguments (Part, in this example) are the GROUP BY fields.
- In order to apply such a rule, must have all of Assembly relation available.
- Stratification with respect to use of **< … >** is the usual restriction to deal with this problem; similar to negation.

Evaluation of Datalog Programs

- Repeated inferences: When recursive rules are repeatedly applied in the naïve way, we make the same inferences in several iterations.
- Unnecessary inferences: Also, if we just want to find the components of a particular part, say wheel, computing the fixpoint of the Comp program and then selecting tuples with wheel in the first column is wasteful, in that we compute many irrelevant facts.

Avoiding Repeated Inferences

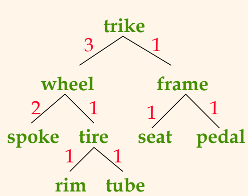
- Seminaive Fixpoint Evaluation: Avoid repeated inferences by ensuring that when a rule is applied, at least one of the body facts was generated in the most recent iteration. (Which means this inference could not have been carried out in earlier iterations.)
  - For each recursive table P, use a table delta_P to store the P tuples generated in the previous iteration.
  - Rewrite the program to use the delta tables, and update the delta tables between iterations.

Comp(Part, Subpt) :- Assembly(Part, Part2, Qty),
delta_Comp(Part2, Subpt).

Avoiding Unnecessary Inferences

SameLev(S1,S2) :- Assembly(P1,S1,Q1), Assembly(P2,S2,Q2).
SameLev(S1,S2) :- Assembly(P1,S1,Q1),
SameLev(P1,P2), Assembly(P2,S2,Q2).

- There is a tuple (S1,S2) in SameLev if there is a path up from S1 to some node and down to S2 with the same number of up and down edges.



Avoiding Unnecessary Inferences

- Suppose that we want to find all SameLev tuples with spoke in the first column. We should "push" this selection into the fixpoint computation to avoid unnecessary inferences.
- But we can't just compute SameLev tuples with spoke in the first column, because some other SameLev tuples are needed to compute all such tuples:

SameLev(spoke,seat) :- Assembly(wheel,spoke,2),
<u>SameLev(wheel,frame)</u>, Assembly(frame,seat,1).

"Magic Sets" Idea

- Idea: Define a "filter" table that computes all relevant values, and restrict the computation of SameLev to infer only tuples with a relevant value in the first column.

Magic_SL(P1) :- Magic_SL(S1), Assembly(P1,S1,Q1).
Magic(spoke).

SameLev(S1,S2) :- Magic_SL(S1), Assembly(P1,S1,Q1),
Assembly(P2,S2,Q2).
SameLev(S1,S2) :- Magic_SL(S1), Assembly(P1,S1,Q1),
SameLev(P1,P2), Assembly(P2,S2,Q2).

The Magic Sets Algorithm

- Generate an "adorned" program
  - Program is rewritten to make the pattern of bound and free arguments in the query explicit; evaluating SameLevel with the first argument bound to a constant is quite different from evaluating it with the second argument bound
  - This step was omitted for simplicity in previous slide
- Add filters of the form "Magic_P" to each rule in the adorned program that defines a predicate P to restrict these rules
- Define new rules to define the filter tables of the form Magic_P

Generating Adorned Rules

- The adorned program for the query pattern SameLev^{bf}, *assuming a right-to-left order of rule evaluation* :

SameLev^{bf} (S1,S2) :- Assembly(P1,S1,Q1), Assembly(P2,S2,Q2).

SameLev^{bf} (S1,S2) :- Assembly(P1,S1,Q1),
SameLev^{bf} (P1,P2), Assembly(P2,S2,Q2).

- An argument of (a given body occurrence of) SameLev is b if it appears to the left in the body, or in a b arg of the head of the rule.
- Assembly is not adorned because it is an explicitly stored table.

Defining Magic Tables

- After modifying each rule in the adorned program by adding filter "Magic" predicates, a rule for Magic_P is generated from each occurrence O of P in the body of such a rule:
  - Delete everything to the right of O
  - Add the prefix "Magic" and delete the free columns of O
  - Move O, with these changes, into the head of the rule

SameLev^{bf} (S1,S2) :- Magic_SL(S1), Assembly(P1,S1,Q1),
SameLev^{bf} (P1,P2), Assembly(P2,S2,Q2).

Magic_SL(P1) :- Magic_SL(S1), Assembly(P1,S1,Q1).

Summary

- Adding recursion extends relational algebra and SQL-92 in a fundamental way; included in SQL:1999, though not the core subset.
- Semantics based on iterative fixpoint evaluation. Programs with negation are restricted to be stratified to ensure that semantics is intuitive and unambiguous.
- Evaluation must avoid repeated and unnecessary inferences.
  - "Seminaive" fixpoint evaluation
  - "Magic Sets" query transformation