ECS 165B: Database System Implementation
Lecture 25

UC Davis
May 24, 2010

Acknowledgments: some slides due to Ramakrishnan and Gehrke

Class Agenda

- Last time:
  - Query Evaluation Engine Cookbook Session
  - Overview of Column Stores

- Today:
  - Deductive Databases

- Reading:
  - Chapter 24 of Ramakrishnan and Gehrke
    (Section 4.7 of Silberschatz et al)

Deductive Databases

Motivation

- SQL, as we've seen it so far, 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?
- (Aside: how can you *prove* such statements?)
  - Using tools from **finite model theory**, such as Ehrenfeucht–Fraïssé games (ECS 289F)
- Can we extend SQL to cover such queries?
  - Yes, by adding recursion...

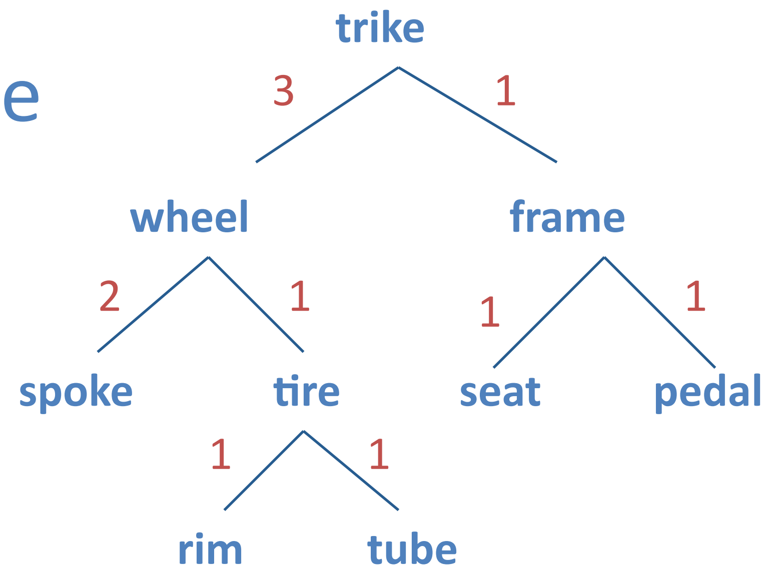
Datalog

- 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 toy 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



Assembly

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

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

The Problem with RA 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 RA 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

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

- Each rule can be viewed as a **template:** by assigning constants to the variables in such a way that each atom in body 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: Deducing New Tuples

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

- For any instance of `Assembly`, we can compute all `Comp` tuples by repeatedly applying the two rules

Comp tuples after applying rules once:

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

Comp tuples after applying rules twice:

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

Datalog versus SQL Notation

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

```
WITH RECURSIVE Comp(Part, Subpt) AS (
    (SELECT Part, Subpt
     FROM Assembly)
    UNION
    (SELECT A.Part, C.Subpt
     FROM Assembly A, Comp C
     WHERE A.Subpt=C.Part)
)
SELECT Part, Subpt FROM Comp
```

- Current commercial DBMSs support a limited amount of recursive queries, via syntax like above

Defining the Semantics: Fixpoints

- **Definition**: Let *f* : *D* → *D*. A value *v* in *D* is a fixpoint of *f* if *f*(*v*)=*v*.
- **Example 1**: consider the function *double* from integers to integers which multiplies its argument by 2. Then 0 is a fixpoint of *double* (in fact, the only fixpoint).
- **Example 2**: consider a function *double+*, which is applied to a **set** of integers and returns a **set** of integers, and works like: *double+*({1,2,5}) = {2,4,10} U {1,2,5} = {1,2,4,5,10} . Then
  - 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

- **Definition**: the **least fixpoint** of a function *f* is a fixpoint *v* of *f* such that every other fixpoint of *f* is ≤ *v*.
- In general, there may be no least fixpoint (we could have no fixpoint, or 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.

Aside: Other Ways of Defining Datalog's Semantics

- Besides the least fixpoint semantics, datalog can be defined in two other ways:
  - **proof-theoretic**: a tuple is in the answer iff it can be "proven" using the source database and the rules of the program
  - **model-theoretic**: view the rules as a collection of logical assertions; the result of the program is the smallest *model*, where a *model* is a database instance (including both source and derived relations) that satisfies the assertions
- These turn out to be equivalent to the fixpoint-theoretic semantics!

Extending Datalog with 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!

The Simplest Fix: Stratification

- T depends on S if some rule with T in the head contains S or (recursively) some predicate that depends on S, in the body.
- Stratified program: 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 source 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 Programs

- The semantics of a stratified program is given by one of the minimal fixpoints, which is identified by the following operational definition:
  - 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 (considering Stratum 1 as "source tables"); then the lfp of tables in Stratum 3, and so on, stratum-by-stratum.
- Note that `Big/Small` program is not stratified.

Aside: Beyond Stratified Semantics

- Not all programs are stratified; can we give semantics to those too?
- Yes, using e.g., *stable model semantics*, or the *well-founded semantics*
- Cool topics, but beyond the scope of what we're covering
- Prof. Ludaescher did some of the seminal work on the latter (well-founded semantics) as a PhD student

Complexity of Datalog with Stratified Semantics

- In databases, have to distinguish between two kinds of complexity: *data complexity* and *query complexity*
  - data complexity: query is fixed, database may vary in size
  - query complexity: database is fixed, query may vary in size
  - (combined complexity: both may vary in size)
- Queries are small in practice, hence data complexity is the one we worry about most
- Fact: can evaluate stratified Datalog programs in polynomial time (data complexity)

P=NP? is a Database Theory Problem

- Here's a mind-blowing result from a field now known as *descriptive complexity:*
- Suppose you have a total order < on the underlying domain of the database, and can use < in queries
  - goes without saying in practical applications, but the logicians don't take this at all for granted
- Theorem [Vardi]: Datalog with stratified semantics with < *captures* the polynomial-time computable queries
- So, to show P != NP, "just" have to prove that SAT is not expressible in Datalog!

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).
```

- Just like "delta rules" technique for incremental view maintenance