INFO2820 – Database Systems I (Adv)

Week 3: DATALOG with Recursion and Negation
(Kifer/Bernstein/Lewis – Chapter 13.6; Ullman/Widom – Chapter 5.3)

Dr. Uwe Röhm
School of Information Technologies

Outline

- Recursion in Datalog
- Naïve non-negated evaluation
  - Proof of completion
  - Complexity estimation
- Datalog with Negation and Recursion
  - Stratification Algorithm

Based on material from Kifer/Bernstein/Lewis (2006) “Database Systems”

Evaluating Datalog Programs

- A ***Datalog program*** is a collection of rules.
- In a program, predicates can be either
  - **EDB** = *Extensional Database* = stored table.
  - **IDB** = *Intensional Database* = relation defined by rules.
- Evaluation Strategy:
  - As long as there is no recursion, we can pick an order to evaluate the IDB predicates, so that all the predicates in the body of its rules have already been evaluated.
  - If an IDB predicate has more than one rule, each rule contributes tuples to its relation.
  - But what about recursion?

Recursive Datalog Rules

- Example (from last week's tutorial):
- EDB:
  assembly (Part, Subpart, Count)
- IDB:
  component(Part, Subpart) :- assembly(Part, Subpart, _).
  component(Part, Subpart) :- assembly(Part, Part2, _),
  component(Part2, Subpart).
- Finds the ***transitive closure*** of the trike components.
- Note how the 'component' predicate appears both in the head and the body of the rules.

Dependency Graph and Recursion

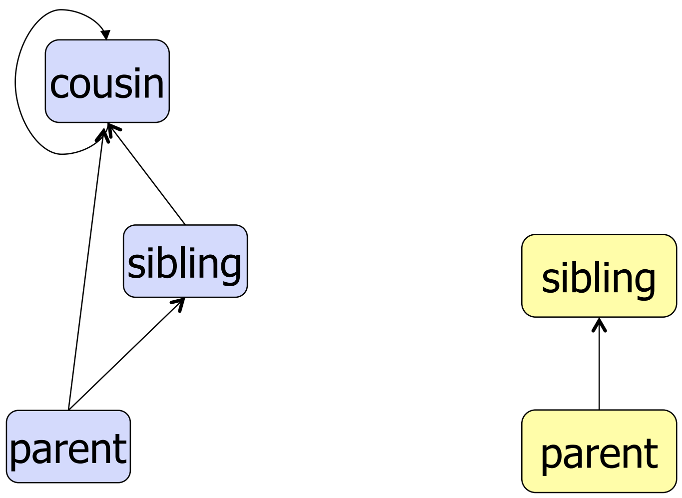
- A ***dependency graph*** is a graph that models the way that predicates depend on themselves.
- Given Datalog program *P*, the dependency graph of *P* has:
  - a node for each predicate in *P*
  - an edge from a predicate *x* to a predicate *y* if there is a rule with *y* in the head and *x* in the body
- A predicate *R* is **recursive** if there is a cycle in its dependency graph
  - conversely:
    if the dependency graph is acyclic, then program *P* is not recursive

Another Recursive Example

- EDB: parent(*P,C*) = *P* is a parent of *C*.
- IDB: Generalized cousins
  people with common ancestors one or more generations back

```
sibling(X,Y) :- parent(P,X) , parent(P,Y) , X<>Y.
cousin(X,Y) :- sibling(X,Y).
cousin(X,Y) :- parent(Px,X) , parent(Py,Y) , cousin(Px,Py).
```

Example: Dependency Graphs

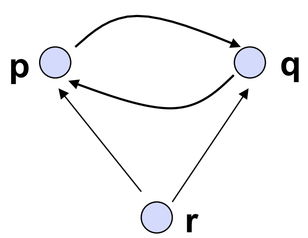


Recursive

Nonrecursive

Example (2)

p(X):- r(X), q(X)
q(X):- r(X), p(X)



- Which predicates are recursive?
- What does this program compute?

Not all Datalog Programs with a cyclic dependency graph are inherently recursive

- Example:

  buys(X, Y) :- trendy(X), buys(Z, Y).
  buys(X, Y) :- likes(X, Y).

  - (person X buys product Y if X likes Y or if X is trendy and someone buys Y)

- Program is equivalent to the following:

  buys(X, Y) :- trendy(X), likes(Z, Y).
  buys(X, Y) :- likes(X, Y).

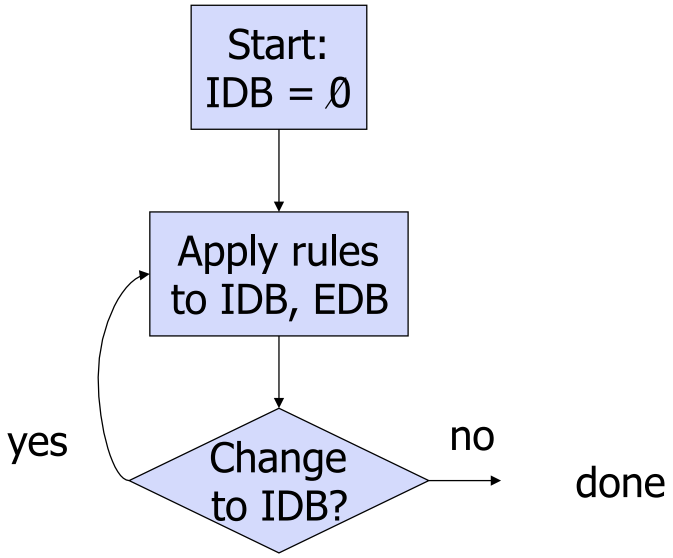
- The following program is inherently recursive:

  buys(X, Y) :- knows(X, Z), buys(Z, Y).
  buys(X, Y) :- likes(X, Y)

Naïve Evaluation Algorithm for Recursive Datalog Queries w/out Negation

- Initialise *CurrentState*
  - All *base relations* are set to instances from the current database
  - all *derived relations* = {}
  - *NewState* = *CurrentState*
- **for each** derived relation ***R***:
  - *answer* = {}
  - For each rule r_i that has *R* in the head
    - Evaluate the query that corresponds to rule r_i
    - *answer* = *answer* U *result*(r_i)
  - *NewState* = *NewState* U *answer*
- **if** *CurrentState* = *NewState* **then** stop
  else *CurrentState* = *NewState* ; repeat step 2

The “Naïve” Evaluation Algorithm



Properties of this Algorithm

- Algorithm always terminates
- It has a polynomial runtime

- Such are important properties for any algorithm…
- … but how do we proof these properties for the given one?

Proof of Termination

- Assumption: All Datalog rules have the following form
  HeadRelation = { *HeadVars* | ∃ *BodyOnlyVars* (*RuleBody*) }
  - In particular: no negation and no all quantification!
- Lemma: Invariant is *CurrentState* ⊆ *NewState*
  - This means that *CurrentState* and *NewState* grow monotonically
  - Initial: *CurrentState* is initialised with the set of relational instances **db0** and *NewState* is assigned *CurrentState;* all derived relations={}
    - *CurrentState* = *NewState* => *CurrentState* ⊆ *NewState* holds
- Key observation: state cannot grow indefinitely
  - Initial db state can only contain a finite number of constants
  - Relations have fixed arity => there can only be a finite number of tuples that can be generated from a given finite set of constants
  - Hence growth of *CurrentState* and *NewState* will eventually stop

Proof of Termination (cont'd)

- Observe that if **db** and **db'** are sets of relation instances for the relations used in all rule bodies, and if **db** ⊆ **db'**, then the queries will produce more results on **db'** than on **db**
- Proof of monotonicity:
  - All rule bodies are conjunctions of predicates $\boldsymbol{P}(X_1, \ldots, X_m)$ or comparisons
  - $N = 1$: just one conjunct: either a relation test or a comparison.
    - The larger the base relation, the larger the result of $P(X_1, \ldots, X_m)$
    - Hence monotonicity rules for case N=1
  - *Induction step i -> i+1:* $RuleBody_{i+1} = RuleBody_i$ AND *Conjunct*
    - Induction assumption: monotonicity holds for *i* conjuncts in $RuleBody_i$
    - If *Conjunct* is a comparison: result of $RuleBody_{i+1}$ is a selection on $RuleBody_i$ which is monotonic
    - If *Conjunct* is a predicate of the form $P(X_1, \ldots, X_m)$ then the result of $RuleBody_{i+1}$ is a join of the result of $RuleBody_i$ and the mootonic query *P* which is monotonic
  - Hence all positive, non-recursive Datalog queries are monotonic

Complexity Estimation

- We can also show that the naïve algorithm will terminate after a polynomial number of steps in the size of the database:

- Let D be the number of constants in the database and N be the sum of arities of all relations used in the query.
- Then each iteration of the algorithm cannot take more than D^N steps
  - This is the cost of producing the Cartesian product of all attribute domains involved in the query
- Since each iteration must produce at least one new tuple and there cannot be more than D^N tuples in the result, the computation must stop after D^N iterations (worst case).
- Thus the number of steps is bounded by D^N x D^N

- If we fix the query, N is constant. Therefore only D depends on the size of the database.

=> Hence the number of steps is polynomial in the size of the database

Negation

- Negation in Datalog gives us two main problems:
- Careful formulation of negative rules that model the correct scenario
  - Note that the Naïve datalog evaluation algorithm is not applicable because with negation in the rule body, the mapping is no longer monotonic…
    - unLucky(D) :- likes(D,Beer), frequents(D,'rose'), NOT sells('rose',Beer,_)
  - hence there could be cases where the algorithm does not terminate
- Negation and recursion
  - Can lead to circular reasoning arguments…

Dependency Graph

- A dependency graph *D = (V, E)* is a set of nodes *V* that are connected with directed edges *E* such that
  - The nodes *V* of the graph are names of EDB or IDB relations
  - If relation *P* occurs in the head and *Q* in the body of the same rule then the graph has a directed *edge* leading from Q to *P*.
  - If *Q* occurs under the scope of a **not** in the rule body, then the edgeis labeled with a '-' sign - a *negative arc.* Otherwise, the edge is unlabeled and called *positive arc.*

- An edge in *D* from Q to *P* means that computing some tuples in *P* requires computing some tuples in Q first
  - *some*, not *all*, otherwise no recursive rules possible
- If the edge is negative, computing *P* requires knowing some tuples in the *complement of* Q
  - to know the *complement of* Q, the *entire* relation Q must be computed first
  - hence no circular negative arcs are allowed

Example Dependency Graph

- Example:

  cheap(Beer, Bar) :- sells(Bar,Beer,P), P < 4.00.
  happy(D) :- likes(D,Beer), cheap(Beer, _).
  unLucky(D) :- likes(D,Beer), frequents(D, Bar), NOT cheap(Beer, Bar).

Computing Datalog Queries with Negative Cycles

- 1. Partition
  - Split the dependency graph into *positively-strongly connected components*, so-called **strata.**
  - **Each stratum consists of a maximal set of nodes such that**
    - There are no negative arcs connecting *any* pair of nodes in the set.
    - For *every* pair of nodes in the set, there is a *positive path* connecting them.
- 2. Stratify
  - Order the strata: if there is a path from some node in stratum S1 to a node in another startum S2, then S1 must proceed S2.
    - Gives a partial order
    - Any *total order* of the strata that is consistent with this partial order is called ***stratification***
- 3. Evaluate
  - Evaluate the strata in the order of the stratification using the algorithm for computing positive recursive queries
    - Replace negated relations (will be computed in previous step) with complement

Properties of Stratification Algorithm

- Existance of stratification
  - If a dependency graph has no negative cycles, a stratification must exist
  - Proof by contradiction: Assume no negative cycle, but also no stratification
    - Suppose, there is a path from a node $n \in$ A to a node $n' \in$ B, and back from a node $m' \in$ B to $m \in$ A with A and B two different strata.
    - Since *m* and *n* belong to the same stratum, there must be a path connecting *m* and *n* in A; The same holds for m' and n' in B
    - Hence there must be a cycle between these four nodes.
    - Since there is no negative cycle, this cycle must be positive.
    - But then, by the definition of stratum, all node of A U B must be in the same stratum - contradicting the assumption.
  - Hence there is no positice cycle between nodes in a dependency graph
  - Let $A < B$ mean that there is a path from a node in A to a node in B in the dependency graph. The absence of cycles means that < is a partial order.
    - Every partial order can be extended to a total order => stratification
- Do all stratifications of a query yield the same result?
  - there can be more than one total ordering for a partial order…
  - Answer: Yes [Apt et al 1988]

You should now be able to:

- understand the execution model of Datalog
- understand simple termination and complexity proofs
- find the dependency graph for a given set of Datalog rules
- explain stratification of Datalog rules, and the core problems with negation and recursion in Datalog

References

- Kifer/Bernstein/Lewis (2nd edition)
  - Chapter 13.6
  - *The proofs and the stratification explanations are from this Chapter.*
- Ramakrishnan/Gehrke (3rd edition - the 'Cow' book)
  - Chapter 24
- Ullman/Widom (3rd edition)
  - Chapter 5.4
  - *No proofs/stratification but good introduction to Datalog & how it relates to RA.*
- Garcia-Molina/Ullman/Widom (1st edition)
  - Chapter 10
- Silberschatz/Korth/Sudarshan (5th edition - 'sailing boat')
  - Chapter 5.4
  - *Good overview of Datalog and recursion, but nothing on stratification.*
- Elmasri/Navathe (5th edition)
  - Chapter 24.4
  - *Datalog + introduces predicate dependency graph, but no stratification algo.*