CS 784: Foundations of Data Management *Instructor: Paris Koutris*

Lecture 9: Datalog: Evaluation

In this lecture we will focus on how we can evaluate Datalog programs efficiently. We will examine two different techniques: top-down and bottom-up evaluation. First, let's write another example of a Datalog program, which we will use as a running example.

Example 1

Write a Datalog program that, given a graph with edge relation $R(x, y)$, computes all pairs of nodes (x, y) such that there is a directed path of even (odd) length from x too y.

```
Odd(x,y)  :- R(x,y).
Even(x,y) :- Odd(x,z), R(z,y).
Odd(x,y)  :- Even(x,z), R(z,y).
```

9.1 Naive Evaluation and Complexity

We start with the simplest bottom-up approach to evaluate a Datalog program P, called *naive evaluation*. Let $P_1, \ldots, P_m$ be the intensional relations (IDBs) of the program P. For every relation P_i, let q_i be the *immediate consequence operator*, in other words a single application of the rules that contain P_i as the head. Notice that q_i is a UCQ, since it can be computed as a union of the rules, where each rule is a conjunction of predicates followed by a projection.

For our example, the query q_{Odd} that computes the IDB *Odd* is:

$$q_{Odd} = R(x, y) \cup \pi_{x,y}(Even(x, z) \bowtie R(z, y)).$$

Similarly, the query q_{Even} that computes the idb *Even* is:

$$q_{Even} = \pi_{x,y}(Odd(x, z) \bowtie R(z, y)).$$

Notice that the intensional relations of the Datalog program depend on each other, in which case we call the program *mutually recursive*.

The naive algorithm iteratively computes versions of each predicate $P_i^{(k)}$ by applying the immediate consequence operator q_i on the previous versions of these predicates.

1. for every IDB P_i : $P_i^{(0)} \leftarrow \emptyset$
2. $k \leftarrow 0$

3. **repeat**
   - $k \leftarrow k+1$
   - For every IDB P_i: $P_i^{(k)} \leftarrow q_i(P_1^{(k-1)}, \ldots, P_m^{(k-1)})$

   until for every i, $P_i^{(k)} = P_i^{(k-1)}$
4. **output** $P_1^{(k)}, \ldots, P_m^{(k)}$

The number of iterations of the naive algorithm is polynomial in the size of the input, since the total number of facts that can be produced is polynomial, and at each iteration we add at least one new fact (otherwise the algorithm terminates). Moreover, since at each iteration we compute a number of UCQs in parallel, each iteration needs polynomial time. Thus, the data complexity for evaluating Datalog is P. In fact, it is also P-complete.

Theorem 1: Complexity of Datalog

The data complexity of Datalog is P-complete.

We will not show this here, but the *combined complexity* of Datalog is EXP-complete (this is a complexity class that is higher than PSPACE).

9.2 Semi-Naive Evaluation

Let's look again at the Datalog program that computes the transitive closure of a graph.

```
T(x,y) :- R(x,y).
T(x,y) :- T(x,z), T(z,y).
```

Consider the naive evaluation of the above program for the case where the graph is a path of length n: $\{R(1,2), R(2,3), \ldots, R(n-1,n)\}$. Notice that the fact $T(1,3)$ is discovered at the second iteration, as a result of joining the tuples $T(1,2), T(2,3)$. However, $T(1,3)$ will also be discovered in exactly the same way at every subsequent iteration. This is redundant computation, and to overcome this problem we will consider a different type of bottom-up evaluation, called *semi-naive evaluation*. Semi-naive evaluation tries to to compute only new facts and reduce the number of times the old fact are rediscovered.

The idea behind semi-naive evaluation is the following. Suppose we want to compute the rule: $T^{(i+1)}(x,y) := q(T^{(i)}, T^{(i)})$, where $q(R,S) = \pi_{x,y}(R(x,z) \bowtie S(z,y))$. We write $T^{(i)}$ as $T^{(i-1)} \cup \Delta T^{(i)}$, where $\Delta T^{(i)}$ are the *new tuples* discovered at the i-th iteration. Now we can write:

$$\begin{aligned} T^{(i+1)} &= q(T^{(i-1)} \cup \Delta T^{(i)}, T^{(i-1)} \cup \Delta T^{(i)}) \\ &= q(T^{(i-1)}, T^{(i-1)}) \cup q(T^{(i-1)}, \Delta T^{(i)}) \cup q(\Delta T^{(i)}, T^{(i-1)}) \cup q(\Delta T^{(i)}, \Delta T^{(i)}) \\ &= T^{(i)} \cup q(T^{(i-1)}, \Delta T^{(i)}) \cup q(\Delta T^{(i)}, T^{(i-1)}) \cup q(\Delta T^{(i)}, \Delta T^{(i)}) \end{aligned}$$

The above rewriting of the query was a simple distribution of union over the join operator. Now, instead of evaluating the query from scratch, we can *incrementally* compute the fresh tuples by joining the newly discovered tuples with the previous ones. This is exactly how incremental view maintenance also works in relational databases! For a UCQ query q, write Δq to denote its incremental evaluation. We can now write the algorithm for semi-naive evaluation:

1. For all IDB P_i : $P_i^{(0)} \leftarrow \emptyset$
2. $k \leftarrow 1$
3. In parallel do $\Delta P_i^{(1)} \leftarrow q_i(P_1^{(0)}, \ldots, P_m^{(0)})$
4. **repeat**:
   - $k \leftarrow k + 1$
   - $P_i^{(k-1)} \leftarrow P_i^{(k-2)} \cup \Delta P_i^{(k-1)}$
   - In parallel do $\Delta P_i^{(k)} \leftarrow \Delta q_i(P_1^{(k-2)}, \ldots, P_m^{(k-2)}, \Delta P_1^{(k-1)}, \ldots, \Delta P_m^{(k-1)}) - P_i^{(k-1)}$

   until for every $i, \Delta P_i^{(k)} = \emptyset$
5. **output** $P_1^{(k)}, \ldots, P_m^{(k)}$

There are some Datalog programs where semi-naive evaluation becomes extremely effective.

Definition 1

A rule in a Datalog program is *linear* if there is at most one atom in the body of the rule whose predicate is mutually recursive with the head of the rule.

For example, the rule $T(x,y) : -T(x,z), R(z,y)$ is a linear rule. In this case, the incremental operator becomes: $\Delta T^{(k)} = \pi_{x,y}(\Delta T^{(k-1)} \bowtie R) - T^{(k-1)}$; in other words, to compute the new tuples we only need to use the new tuples from the previous iteration, and not the rest of the intensional relation. Hence, linear rules are in general amenable to very efficient semi-naive evaluation.

Another optimization that can be done on top of the above algorithm is to impose some order on how we evaluate the IDB relations. To do this, we construct the *precedence graph*, where the vertices are the IDB relations, and we add an edge (R, S) appears in the body of a rule with head S. We then do a topological sorting of the directed graph, and we evaluate each strongly connected component in the topological order (bottom-up) using the semi-naive algorithm.

9.3 Top-Down Evaluation

Consider the following Datalog program:

```
T(x,y) :- R(x,y).
T(x,y) :- T(x,z), R(z,y).
q(y)   :- T('a',y).
```

Suppose we are only interested in evaluating the predicate q, in other words we want only to compute the nodes that are reachable from node a. The semi-naive evaluation of the above program is very wasteful, since it will first compute all the transitive closure T, and then perform on selection to choose only the paths that start with node a. In this case, the bottom-up approach is not very effective. We present next a top-down algorithm called the *Query-Subquery (QSQ) algorithm*.

The are two key ideas in this algorithm: *binding patterns* and *supplementary relations*.

Binding Patterns. For each idb, we will consider an *adorned* version based on the bindings of the variables that are considered. For example, because of the rule $q(y) : -T('a', y)$, we are only interested in finding derivations for T where the first coordinate is *bound (b)* and the second coordinate is *free (f)*. To denote this, we will use the adorned idb T^{bf}. Now, if we want to compute the rule $T(x,y) : -T(x,z), R(z,y)$ for T adorned as T^{bf}, since x is bound, we can write the rule as:

$$T^{bf}(x,y) \text{ :- } T^{bf}(x,z), R(z,y).$$

Here we should note that the same idb may appear with different adornments in a Datalog program. However, different adornments of the same relation are treated as different relations during the computation! The algorithm for computing adornments of a rule is given below:

1. All occurrences of each bound variable in the rule head are bound.
2. All occurrences of constants are bound.
3. If a variable x occurs in the body, then all occurrences of x in subsequent atoms are bound.

Supplementary Relations. For each adorned idb and each position in the body of a rule, we define a *supplementary relation* that accumulates the bindings relevant to that position. The first supplementary relation sup_0 has only the bound variables from the head, the last supplementary relation has all the variables in the head, and the intermediate supplementary relations have the bound variables at that position. For example, consider the rule $T^{bf}(x,y) : -T^{bf}(x,z), R(z,y)$. The rule with the supplementary relations will be as follows:

$$T^{bf}(x,y) : -[sup_0(x)]T^{bf}(x,z), [sup_1(x,z)]R(z,y)[sup_2(x,y)]$$

The full example Datalog program will now look as follows:

$$\begin{aligned} T^{bf}(x,y) &\text{ :- } [sup_0^1(x)]R(x,y)[sup_1^1(x,y)]. \\ T^{bf}(x,y) &\text{ :- } [sup_0^2(x)]T^{bf}(x,z), [sup_1^2(x,z)]R(z,y)[sup_2^2(x,y)]. \\ q(y) &\text{ :- } T^{bf}('a', y). \end{aligned}$$

For each adorned relation, we will also consider a corresponding *input relation* with the same arity as the number of bound variables. For example, let $in\_T^{bf}(x)$ be the input relation for T^{bf}. To

evaluate the program using the *query-subquery* algorithm, we start with the output goal q, which gives the initial value a to $in\_T^{bf}(x)$. The input relation then populates the first supplementary relations of each rule $sup_0^1(x), sup_0^2(x)$. The new values are then propagated to the other supplementary relations from left to right. The final supplementary relations then feed the new tuples back to the adorned relation T^{bf}. The process ends when no new facts are produced.

9.4 Magic Sets

The top-down computation allows us to reduce unnecessary computation by disregarding facts that we will not need at all. A surprising fact about Datalog is that we can achieve exactly the same result by bottom-up evaluation techniques through rewriting the Datalog program in a way that is similar to pushing the selection down a relational algebra query plan. This technique is called *magic sets*. The idea of the magic set transformation is to use the adorned relations and supplementary relations in order to simulate pushing down selections.

Below is the rewriting of our running Datalog example. The first rule is rewritten as follows:

$$\begin{aligned} sup_0^1(x) &: -in\_T^{bf}(x). \\ sup_1^1(x,y) &: -sup_0^1(x), R(x,y). \\ T^{bf}(x,y) &: -sup_1^1(x,y). \end{aligned}$$

The second rule:

$$\begin{aligned} sup_0^2(x) &: -in\_T^{bf}(x). \\ sup_1^2(x,z) &: -sup_0^2(x), T^{bf}(x,z). \\ sup_2^2(x,y) &: -sup_1^2(x,z), R(z,y). \\ T^{bf}(x,y) &: -sup_2^2(x,y). \end{aligned}$$

We finally need to initialize the input relations:

$$\begin{aligned} in\_T^{bf}(x) &: -T^{bf}(x,y). \\ in\_T^{bf}('a') &: - \\ q(y) &: -T^{bf}('a',y). \end{aligned}$$

Theorem 2

The set of facts produced when we execute the semi-naive algorithm on the magic-set transformed Datalog program is identical to the set of facts produced by a QSQ evaluation.

References

[Alice] S. ABITEBOUL, R. HULL and V. VIANU, "Foundations of Databases."