Amoeba: A Rust Implementation of Datalog

Yifei Zuo
University of Science and Technology of China
yifeizuo@mail.ustc.edu.cn

I. Introduction

Relational algebra is the core language for database with limited expressibility. A representative example of this is the inability to solve *transitive closure* problems. Datalog is a logic programming language that extends relational algebra with recursion, which allows use to express more complex relationship.

I.1. Datalog Syntax

A *Datalog rule* is an expression of the form:

$$R(\vec{x}) \leftarrow R_1(\vec{x_1}), R_2(\vec{x_2}), ..., R_n(\vec{x_n})$$

where we allow the for a relation R on the left hand side of the rule to appear on the right hand side. A *Datalog program* is a finite set of Datalog rules. A *Datalog program* consists of two kinds of schemas:

- **Extensional Schemas (EDB)** are set of *external relations* that only appear on the right hand side of rules. They are intuitively the initial input data of a Datalog program.
- **Intensional Schemas (IDB)** are set of *internal relations* that appear at least once on the left hand side of rules. They are intuitively the output of a Datalog program.

A Datalog program semantically is a mapping from database instances over the extensional schemas to database instances over the intensional schemas.

Example. *Datalog program expressing the transitive closure of a relation r in* linear rule *style, which allows for the rule on the left appears exactly once on the right hand side of the rule*:

```
t(X, Y) :- r(X, Y).
t(X, Y) :- r(X, Z), t(Z, Y).
```

The following is a nonlinear *Datalog program implementing the same transitive closure*:

```
t(X, Y) :- r(X, Y).
t(X, Y) :- t(X, Z), t(Z, Y).
```

I.2. Datalog Semantics

There are three equivalent ways to define the semantics of a Datalog program: model-theoretic, fixpoint, and proof-theoretic. Here we discuss the model-theoretic semantics and fixpoint semantics.

I.2.1. Model-Theoretic Semantics

Model-theoretic semantics associates a first-order logic interpretation to a Datalog program. For example, the rule $\rho : T(x, y) \leftarrow T(x, z), R(z, y)$ gives the following logical sentence $\Phi_\rho = \forall x, y, z(T(x, z) \wedge R(z, y)) \rightarrow T(x, y)$. Generally, a rule ρ could associate the following logical sentence:

$$\Phi_\rho = \forall x_1, x_2, ..., x_n(R_1() \wedge R_2() \wedge ... \wedge R_n()) \rightarrow R()$$

where $x_1, x_2, ..., x_n$ are the variables in the body of the rule. Note that the logical sentence in above form are *Horn Clauses*, where at most one positive literal appears in the head of the clause. We further assume Σ_P is the set of logical sentences Φ_ρ for every rule ρ in the Datalog program P.

Definition. *Let P be the Datalog program. A pair of instances (I, J) where I is an EDB and J is an IDB, is a model of P if (I, J) satisfies Σ_P. Given an EDB I, the minimal model of P, denote J=P(I), is a minimal IDB J s.t. (I, J) is a model of P.*

An important result of model-theoretic semantics is that the minimal model of a Datalog program always exists and is also unique. Also, the minimal model contains only tuples with values from the active domain **adom(I)**. The semantics of a Datalog program P executed on EDB I is exactly the minimum model P(I).

I.2.2. Fixpoint Semantics

We will be focusing on fixpoint semantics later in this report. Let P be a Datalog program, and an EDB I. For an EDB J, we say a fact or tuple t is an *immediate consequence* of I, J if either $t \in I$ or it is the direct result of a rule application on I, J. We define the *immediate consequence operator* T_P as follows:

Definition. *For every EDB J, $T_{P(J)}$ contains all the facts that are immediate consequence of I, J.*

Lemma. *The operator T_P is monotone, which is, if $I \subset J$ them $T_{P(I)} \subset T_{P(J)}$.*

Definition. *An instance I is a fixpoint for T_P is $T_{P(I)} = I$.*

There's a theorem that states that the minimal model of a Datalog program is the least fixpoint of the immediate consequence operator T_P. This theorem is the basis of the fixpoint semantics of Datalog.

Theorem. *For each Datalog program P and EDB I, the immediate consequence operator T_P has a unique, minimal fixpoint J that contains I, which equals the model $P(I)$.*

The fixpoint semantics gives us an algorithm that computes the output of a Datalog program, which is the main focus of the next section. Generally speaking, we start with the input I, which is the EDB instance, we then compute $T_{P(I)}$, then $T_P^2(I)$ an so on. At some point, after a polynomial number of iterations, we will reach the fixpoint J. The output of the Datalog program is J. This is called the *naive evaluation* algorithm.

II. Evaluation Algorithm

In last section we roughly presented the *naive evaluation*. Typically, Datalog program can be evaluate by bottom-up fashion or top-down fashion. Naive evaluation is a bottom-up algorithm, where we start with the EDB instance and compute the IDB instance like forward chaining that involves huge amount of redundant computation. In fact, a naive evaluation algorithm is exponential in the worst case considering the following:

Theorem. *The data complexity*

- **Bottom-up evaluation** is a forward chaining algorithm that starts with the EDB instance and computes the IDB instance.
- **Top-down evaluation** is a backward chaining algorithm that starts with the IDB instance and computes the EDB instance.

II.1. Semi-Naive Evaluation

In this section we present *semi-naive evaluation* algorithm, which is a more efficient algorithm for computing the output of a Datalog program.

Example. *The following Datalog program defines the Reverse-Same-Generation (RSG)*:

```
rsg(X, Y) :- flat(X, Y).
rsg(X, Y) :- up(X, X1), rsg(Y1, X1), down(Y1, Y).
```

The instances sample is illustrated in Figure 1.

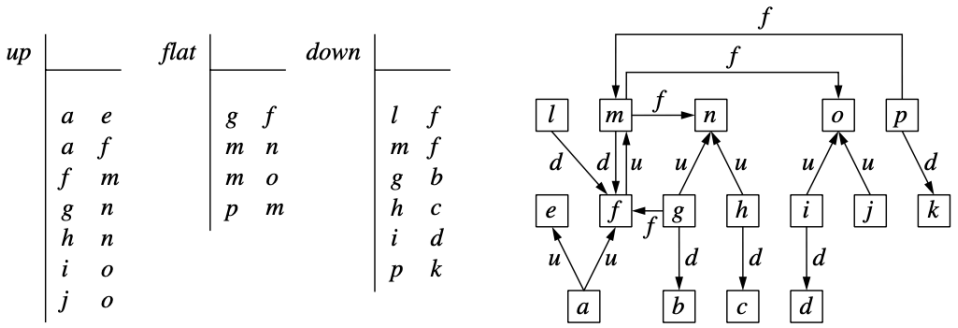

Figure 1: Instance $\boldsymbol{I_0}$ for RSG example

We first try to evaluate the RSG using naive evaluation algorithm on the input instance I_0, the following values are obtained:

level 0: $\emptyset$
level 1: $\{\langle g, f\rangle, \langle m, n\rangle, \langle m, o\rangle, \langle p, m\rangle\}$
level 2: $\{\text{level 1}\} \cup \{\langle a, b\rangle, \langle h, f\rangle, \langle i, f\rangle, \langle j, f\rangle, \langle f, k\rangle\}$
level 3: $\{\text{level 2}\} \cup \{\langle a, c\rangle, \langle a, d\rangle\}$
level 4: {level 3}

at which point a fixpoint has been reached. It is clear that a considerable amount of redundant computation is done, because each layer recomputes all elements of the previous layer. This is a consequence of the monotonicity of the T_P operator. The semi-naive evaluation algorithm avoids this redundant computation by only computing the new tuples that are added to the previous layer.

Consider the facts inferred using the second rule of RSG in the consecutive stages of the naive evaluation. At each stage, some new facts are inferred until a fixpoint is reached. To infer a new fact at stage $i + 1$, one must use at least one new fact newly derived at stage i. This is the main idea of semi-naive evaluation algorithm. It is captured by the following version of RSG, named *delta-RSG*:

```
delta_rsg(X, Y) :- flat(X, Y).
delta_rsg(X, Y) :- up(X, X1), delta_rsg(Y1, X1), down(Y1, Y).
```

where an instance of the second rule is included for each $i \geq 1$. This is essentially not a standard Datalog program because it has an infinite number of rules, and it is not recursive. Intuitively, `delta_rsg` only contains the facts that are newly derived at each stage of the naive evaluation.

We now present the general semi-naive algorithm. Let P be a Datalog program over edb R and idb T. Consider a rule

$$S(u) \leftarrow R_1(v_1), ..., R_n(v_n), T_1(w_1), ..., T_m(w_m)$$

in P, where the R_k's are edb predicates and the T_k's are idb predicates. We construct for each $j \in [1, m]$ and $i \geq 1$ the rule

$$\text{temp}_S^{i+1}(u) \leftarrow R_1(v_1), ..., R_n(v_n), T_1^i(w_1), ..., T_{j-1}^i(w_{j-1}), \Delta_{T_j}^i(w_j), T_{j+1}^{i-1}(w_{j+1}), ..., T_m^{i-1}(w_m)$$

Let P_S^i represent the set of all i-level rules of this form constructed for the idb predicate S (i.e., the rules for $\text{temp}_S^{i+1}, j \in [1, m]$). Suppose now that $T_1, ..., T_l$ is a listing of the idb predicates of P that occur in the body of a rule defining S. We would write

$$P_S^i\left(I, T_1^{i-1}, ..., T_l^{i-1}, T_1^i, ..., T_l^i, \Delta_{T_1}^i, ..., \Delta_{T_l}^i\right)$$

to denote the set of tuples that result from applying the rules in P_S^i to given values for input instance I and for the T_k^{i-1}, T_j^i and $\Delta_{T_j}^i$.

Semi-Naive Evaluation(P, I):

1 Set P' to be the rules in P with no idb predicate in the body;
2 $S^0 := \varnothing$, for each idb predicate S;
3 $\Delta_S^1 := P'(I)(S)$, for each idb predicate S;
4 $i := 1$;
5 **do begin**
6 **for each** idb predicate S **begin**
7 $T_1, ..., T_l$ are the idb predicates in rules defining S;
8 $S^i := S^{i-1} \cup \Delta_S^i$;
9 $\Delta_S^{\{i+1\}} := P_S^i\left(I, T_1^{i-1}, ..., T_l^{i-1}, T_1^i, ..., T_l^i, \Delta_{T_1}^i, ..., \Delta_{T_l}^i\right) - S^i$;
10 **end**
11 $i := i + 1$;
12 **end until** $\Delta_S^i = \varnothing$ for each idb predicate S;
13 $s := s^i$ for each idb predicate S;

In out previous example, the RSG program is transformed into following intermediate representation by applying the semi-naive evaluation:

$$\left\{\begin{array}{c}\Delta_{\text{rsg}}^1(x,y) \leftarrow \text{flat}(x,y) \\ \text{rsg}^1 := \Delta_{\text{rsg}}^1\end{array}\right\}$$

$$\left\{\begin{array}{c}\text{temp}_{\text{rsg}}^{i+1}(x,y) \leftarrow \text{up}(x,\text{y1}) \wedge \Delta_{\text{rsg}}^i(\text{y1},\text{y2}) \wedge \text{down}(\text{y2},y) \\ \Delta_{\text{rsg}}^{i+1} := \text{temp}_{\text{rsg}}^{i+1} - \text{rsg}^i \\ \text{rsg}^{i+1} := \text{rsg}^i \cup \Delta_{\text{rsg}}^{i+1}\end{array}\right\}$$

II.2. Optimization

Above version of semi-naive evaluation algorithm still has a problem doing unnecessary computation. We now analyze the structure of datalog programs to develop an improved version of the semi-naive algorithm. We would like to know in advance which predicates are likely to grow at each iteration and which are not, either because they are already saturated or because they are noe yet affected by the computation.

Let P be a Datalog program. We define the *precedence graph* G_P of P as follows:

Definition. *G_P is the precedence graph* of P, if the nodes are the idb predicates and it includes edge (R, R') if there is a rule with head R' in which R occurs in the body.

Obviously, P is *recursive* if G_P has a directed cycle. Two predicates R' and R are *mutually recursive* if $R = R'$ and R' participate in the same cycle of G_P. Mutual recursion is an equivalence relation on the idb predicates of P, where each equivalence class corresponds to a strongly connected component of G_P. A rule of P is *recursive* if the body involves a predicate that is mutually recursive with the head.

```
IMPROVED SEMI-NAIVE EVALUATION(P, I):
1  Determine the equivalence classes of mutually recursive predicates of idb(P);
2  [R_1], ..., [R_n] are the equivalence classes according to topological sort of G_P;
3  for i = 1 to n begin
4      basic semi-naive algorithm to compute values of predicates in [R_i];
5  end
```

In the for loop of i, we treat all predicates in $[R_j], j < i$ as edb predicates.

In above discussion, we assume no restriction on the Datalog program. In practice, we almost always write *linear* Datalog program which a rule in P with head relation R has at most one occurrence of R in the body. We can further simplify the improved semi-naive algorithm for linear Datalog program.

Suppose that P is a linear program, and

$$\rho : R(u) \leftarrow T_1(v_1), ..., T_n(v_n)$$

is a rule in P, where T_j is mutually recursive with R. Associate this with the "rule"

$$\Delta_R^{i+1}(u) \leftarrow T_1^i(v_1), ..., \Delta_{T_j}^i, ..., T_n^i(v_n)$$

Note that this is the only rule that will be associated by the improved semi-naive algorithm with ρ. Therefore, given an equivalence class $[T_k]$ of mutually recursive predicates of P, the rules for predicates S in $[T_k]$ use only the Δ_S^i, but not the S^i, while both Δ_S^i and S^i are used in non-linear programs.

III. Implementation