Improving Performance of the Datalog Engine in Flix

Zijun Yu, 202203581

Master's Thesis, Computer Science

May 2024

Advisor: Magnus Madsen

AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

Acknowledgments

►. . . ◄

Zijun Yu,
Aarhus, May 2024.

Contents

Chapter 1

Introduction

Datalog is a simple, yet powerful, declarative logic programming language. Programs written in a logic programming language are a collection of logical statements and rules. This allows the programmer to solve problems by stating facts and rules about the problem domain, rather than by giving step-by-step algorithmic instructions.

The Flix programming language is a new functional, declarative, and strongly typed programming language on the JVM. One of the main features is the first-class support for Datalog, allowing the programmers to write Datalog programs directly in Flix.

The Datalog engine in Flix is implemented in Flix as a library, using semi-naïve evaluation as its evaluation strategy as most modern Datalog engines do. The Datalog engine in Flix has two parts. First, it compiles the given Datalog program into an intermediate representation, RAM (Relational Algebra Machine), represented as Flix values. Then it simply interprets the RAM program statement by statement. The semi-naïve evaluation is reflected in the way the RAM program is constructed. The generated RAM program is highly parallelizable, but it requires a concurrent data structure.

The goal of this thesis is to improve the performance of the Datalog engine in Flix. The main contribution of this thesis is the parallelization of the evalution which include three parts: (1) a benchmark suite is proposed for evaluating the performance of the engine, (2) a new parallel statement is added to RAM, (3) a concurrent data structure is implemented to enable the interpretation of the parallel statement. To be more specific, a concurrent B+-tree is implemented to replace the Red-black tree used in the current implementation. The thesis statement is as follows: A concurrent B+-tree would make Datalog evalution in Flix more efficient.

The rest of the thesis is organized as follows. In Chapter 2, we give a brief introduction to Datalog, including the syntax and semantics of the language, as well as the semi-naïve evaluation, and how it leads to parallelism. In chapter 3, we first look at how semi-naïve algorithm is implemented in Flix, that is, how is the RAM program constructed and interpreted. We then look at how parallelism can be introduced. Lastly we touch on the benchmark suite. In chapter 4, we look at the concurrent B+-tree implementation. In chapter 5, we evaluate the performance of the new implementation.

Chapter 2

Background

2.1 Datalog Syntax

A Datalog program consists of *facts*, which are statements that are held to be true, and *rules*, which say how to deduce new facts from known facts. One can think of facts as rows in a relational database table, and rules as queries that can be run on the database, or views that can be materialized.

Example 2.1. The following database stores the edges of a directed graph by means of facts of the form $edge(v_1, v_2)$, meaning that there is an edge from vertex v_1 to vertex v_2 in the graph:

```
edge("a", "b").
edge("b", "c").
edge("c", "d").
```

Below are two rules to compute the transitive closure of a graph stored by means of *edge*-facts:

```
tc(X, Y) :- edge(X, Y).
tc(X, Y) :- edge(X, Z), tc(Z, Y).
```

Intuitively, the first rule above says that if there is an edge from a vertex X to a vertex Y, then (X,Y) belongs to the transitive closure of the graph. The second rule says that if there is an edge from X to Z and there is exists a vertex Y such that (Z,Y) belongs to the transitive closure, then (X,Y) belongs to the transitive closure as well.

We now give the formal syntax of Datalog programs. A *Datalog rule r* is of the form:

$$A_0 : -A_1, \ldots, A_n.$$

where $n \geq 0$, the A_i's are *atoms*, and every atom is of the form $p(t_1, \ldots, t_m)$, where p is a *predicate symbol* and the t_i's are *terms*. A *term* is either a *variable* or a *constant*. In the example above, *edge* and *tc* are predicate symbols, ,"*a*", "*b*", "*c*", "*d*" are constants, and X, Y, Z are variables. Every variable appearing in A_0 (the *head* of a rule) also appears in at least one of the $A_1, \ldots, A_n$ (the *body* of a rule). This requirement is called *safety* and is used to avoid rules yielding infinite sets of facts from a finite set of facts. The intuitive meaning of the Datalog rule above is that if the atoms $A_1, \ldots, A_n$ are true, then the atom A_0 is also true. A *fact* is a rule with an empty body, i.e., a rule of the form $A_0 :-.$ which can be written as A_0..

A *Datalog program* is a finite set of Datalog rules. The *definition* of a predicate symbol p in a program P, denoted $def(p,P)$, is the set of rules of P having p in the head atom. Recall that a database can be seen as a finite set of facts. In the context of logic programming all the knowledge (facts and general rules) is usually contained in a single logic program. We consider two sets of Datalog rules:

1. a set of facts D that represent tuples (rows) of a database; and

2. a Datalog program P whose rules define new realtions (or "views") from the database.

D is called the *Extentional Database* (EDB) and P is called the *Intensional Database* (IDB). We will refer to D as database and P as Datalog program. Thus, predicates symbols are partitioned into two disjoint sets: *base* (or EDB or *extentional*) and *derived* (or IDB or *intensional*). The definition of base predicate symbols is stored in D. Base predicate symbols can appear in the body of rules in P but not in the head. Derived predicate symbols cannot appear in D and their definition is in P. We will use P_D to denote $P \cup D$.

2.2 Datalog Semantics

We give the least-fixpoint semantics of Datalog, which defines the outcome of the program in an operational way, based on repeatedly applying the rules of the program until no new facts can be derived and thus a fixpoint is reached. It is well known that this semantics is equivalent to the model-theoretic semantics as well as the proof-theoretic semantics [1].

The fixpoint semantics is given in terms of an operator called the *immdediate consequence operator* which derives new ground atoms starting from known ground atoms, using the rules of the program. Namely, T_{P_D} takes as input a set of ground atoms I, applies the rules, and returns as output a set of ground atoms $T_{P_D}(I)$, called the *immediate consequences* of I w.r.t. P_D. We say that a set of ground atoms I is a *fixpoint* of T_{P_D} if $T_{P_D}(I) = I$.

It is easy to see that T_{P_D} is monotonic, that is, if $I_1 \subseteq I_2$, then $T_{P_D}(I_1) \subseteq T_{P_D}(I_2)$, for any sets of ground atoms I_1 and I_2. By the Knaster-Tarski theorem, since T_{P_D} is monotonic it has a least fixpoint (that is, a fixpoint that is included in any other fixpoint), which we denote as $lfp(T_{P_D})$. The fixpoint semantics of P_D is given by the least fixpoint $lfp(T_{P_D})$.

Example 2.2. Consider the Datalog program in example 2.1. Applying T_{P_D} yields the following:

$$\begin{aligned}
&I_1 = T_{P_D}(\emptyset) = \{edge(\text{"}a\text{"},\text{"}b\text{"}), edge(\text{"}b\text{"},\text{"}c\text{"}), edge(\text{"}c\text{"},\text{"}d\text{"})\},\\
&I_2 = T_{P_D}(I_1) = I_1 \cup \{tc(\text{"}a\text{"},\text{"}b\text{"}), tc(\text{"}b\text{"},\text{"}c\text{"}), tc(\text{"}c\text{"},\text{"}d\text{"})\},\\
&I_3 = T_{P_D}(I_2) = I_2 \cup \{tc(\text{"}a\text{"},\text{"}c\text{"}), tc(\text{"}b\text{"},\text{"}d\text{"})\},\\
&I_4 = T_{P_D}(I_3) = I_3 \cup \{tc(\text{"}a\text{"},\text{"}d\text{"})\},\\
&I_5 = T_{P_D}(I_4) = I_4.
\end{aligned}$$

Thus, I_4 is the least fixpoint of T_{P_D}.

2.3 Datalog Evaluation

The fixpoint semantics immdediately leads to an algorithm to evaluate Datalog programs, called *naïve*. To define the algorithm, we first define a function called *Eval-Rule* that takes as input a rule r and a set of ground atoms I containing the relations of the predicates appearing in the body of the rule, and returns the set of ground atoms that are the immediate consequences of I w.r.t. r. The function *Eval-Rule* can be defined precisely using relational algebra, but we omit the details here, as the intuition is simple and clear.

Example 2.3. Let r be the second rule in example 2.1 and $I = I_3$ in example 2.2, calling *Eval-Rule* on rule r and relation *edge* and *tc* yields the following:

$$\textit{Eval-Rule}(r, edge, tc) = \{tc(\text{"}a\text{"},\text{"}c\text{"}), tc(\text{"}b\text{"},\text{"}d\text{"}), tc(\text{"}a\text{"},\text{"}d\text{"})\}.$$

We then define another function called *Eval* that evaluate a set of rules whose predicates in the head are the same. Consider a program P and a database D. Let $R_1,\dots,R_n$ be the base relations and $P_1,\dots,P_m$ be the derived relations. For each derived predicate symbol p_i, we define $Eval(p_i, R_1,\dots,R_n, P_1,\dots,P_m)$ as the union of calling *Eval-Rule* over all rules with p_i in the head.

Now, given a datalog program P and a database D, where the base predicate symbols are $r_1,\dots,r_n$, the derived predicate symbols are $p_1,\dots,p_m$, and the base relations are $R_1,\dots,R_n$, algorithm 1 performs the evaluation of P over D.

Algorithm 1 Naive-Evaluation

```
1: for $i := 1$ to $m$ do
2:     $P_i := \emptyset$
3: end for
4: repeat
5:     for $i := 1$ to $m$ do
6:         $P'_i := P_i$
7:     end for
8:     for $i := 1$ to $m$ do
9:         $P_i := \text{Eval}(p_i, R_1, \dots, R_k, P'_1, \dots, P'_m)$
10:    end for
11: until $P_i = P'_i$ for all $1 \le i \le m$
12: return $P_1, \dots, P_m$
```

One shortcoming of the naïve evaluation is that at each iteration, all tuples computed in the previous iterations are recomputed. For example, in example 2.3, the tuples $tc(\text{"}a\text{"},\text{"}c\text{"})$ and $tc(\text{"}b\text{"},\text{"}d\text{"})$ are already computed in the previous iteration (they are already in I_3). The problem is that in the naïve evaluation, in each iteration, the tuples of the derived relations from earlier iterations are used in *Eval-Rule*. The solution to this problem is called *semi-naïve evaluation*.

We start by introducing an incremental version of the *Eval-Rule* function. Consider a rule r of the form

$$A_0 : -A_1, \ldots, A_n.$$

and assume to have a relation R_i and an "incremental" relation ΔR_i for each atom A_i in the body of the rule. We define the incremental version of *Eval-Rule* as follows:

$$\mathit{Eval\text{-}Rule\text{-}Incr}(r, R_1, \ldots, R_n, \Delta R_1, \ldots, \Delta R_n) =$$
$$\bigcup_{1 \le i \le n} \mathit{Eval\text{-}Rule}(r, R_1, \ldots, R_{i-1}, \Delta R_i, R_{i+1}, \ldots, R_n).$$

We can now define the *semi-naïve evaluation* as shown in algorithm 2.

Algorithm 2 Seminaive-Evaluation

```
1:  for $i \leftarrow 1$ to $m$ do
2:      $\Delta P_i \leftarrow$ Eval-rule$(p_i, R_1, \ldots, R_k, \emptyset, \ldots, \emptyset)$
3:      $P_i \leftarrow \Delta P_i$
4:  end for
5:  repeat
6:      for $i \leftarrow 1$ to $m$ do
7:          $\Delta P'_i \leftarrow \Delta P_i$
8:      end for
9:      for $i \leftarrow 1$ to $m$ do
10:         $\Delta P_i \leftarrow$ Eval-rule-incr$(p_i, R_1, \ldots, R_k, P_1, \ldots, P_m, \Delta P'_1, \ldots, \Delta P'_m)$
11:         $\Delta P_i \leftarrow \Delta P_i - P_i$
12:     end for
13:     for $i \leftarrow 1$ to $m$ do
14:         $P_i \leftarrow P_i \cup \Delta P_i$
15:     end for
16: until $\Delta P_i = \emptyset$ for all $1 \le i \le m$
17: return $P_1, \ldots, P_m$
```

It is obvious that the semi-naïve evaluation is parallelizable. There are two level of parallelism in algorithm 2. First, each invocation of *Eval-Rule-Incr* can be done in parallel, that is, the body of loop from line 9 to line 12. Second, as per the definition of *Eval-Rule-Incr*, the invocations of *Eval-Rule* called by *Eval-Rule-Incr* can also be done in parallel. This is the main idea of the parallelization of this thesis. Also, the body of loop from line 1 to line 4 can be done in parallel.

2.4 Datalog extensions and Datalog in Flix

Chapter 3

Datalog Evaluation in Flix

In Flix, the Datalog engine is implemented as a library. When a flix program is executed, the Datalog rules are compiled into an intermediate representation called RAM (Relational Algebra Machine), which is then interpreted statement by statement. The RAM program is constructed in a way that reflects the semi-naïve evaluation algorithm. In this chapter, we first look at how semi-naïve algorithm is implemented in Flix, that is, how is the RAM program constructed and interpreted. We then look at how parallelism can be introduced. Lastly we touch on the benchmark suite.

3.1 RAM

We will not introduce the definition of RAM as flix code here. Instead, we will give examples of stringified RAM programs. We will denote $P_i, \Delta P_i, \Delta P_i'$ in algorithm 2 as *Full* relations, *Delta* relations, and *New* relations.

Example 3.1. Consider the Datalog program in example 2.1. We have one base predicate *edge* and one derived predicate *tc*. The initializations part of the semi-naïve evaluation algorithm (line 1 to line 4) will be translated into the following RAM program:

```
// tc(x, y) :- edge(x, y).
search 0$ ∈ edge do
    project (0$[0], 0$[1]) into Δtc
end;
// tc(x, z) :- tc(x, y), edge(y, z).
search 0$ ∈ tc do
    query {1$ ∈ edge | 1$[0] == 0$[1]} do
        project (0$[0], 1$[1]) into Δtc
    end
end;
merge Δtc into tc;
```

The invocation of *Eval-Rule* for the derived predicate *tc* is translated into the above two RAM statements. The translation is on a rule by rule basis and is mechanical. Variables are represented by integers followed by a dollar sign and they hold a tuple of a relation. Each atom in the body is translated into a *search*. The *search* statement is used to iterate over the tuples of a relation. The *query* statement is like *search* but with a filter of the form `x[0]==a, x[1]==b, x[2]==c, ...` The difference between using a *query* and using a *search* with *if*s is that the former is more efficient, as the tuples are stored using trees and query is interpreted using range query on the trees that avoid unnecessary walking over the tree. We will see more about this in chapter 4.

The main loop of the semi-naïve evaluation algorithm (line 5 to line 16) will be translated into the following RAM program:

```
until(Δtc == ∅) do
    purge Δtc';
    // tc(x, y) :- Edge(x, y).
    // tc(x, z) :- tc(x, y), Edge(y, z).
    search 0$ ∈ Δtc do
        query {1$ ∈ Edge | 1$[0] == 0$[1]} do
            if((0$[0], 1$[1]) ∉ tc) then
                project (0$[0], 1$[1]) into Δtc'
            end
        end
    end;
    merge Δtc' into tc;
    Δtc := Δtc'
end
```

In this program, `tc` is the only derived predicate. Calling *Eval-Rule-Incr* for `tc` includes translation of both rules. However, there is no derived predicate in the body of the first rule, so the first rule is translated to an empty statement. For the second rule, the derived predicate `tc` appears in one atom in the body, so the *search* statement is used to iterate over the *New* relation `Δtc`. In algorithm 2, line 11 is used to compute the `New` relations by set difference. In the implementation, it is done by the *if* operation to check if the tuple is already in the `Full` relation.

We will show one more example of a more complex Datalog program, where derived predicates appear more than once in the body of a rule, which will make the translation of *Eval-Rule-Incr* more interesting.

Example 3.2. The following Datalog program perform a point-to analysis of varaibles in a C program [2]. The instructions of the program can be translated into *EDB* relations as follows:

`AddressOf(a, b)` representing `a=\&b`
`Assign(a, b)` representing `a=b`
`Load(a, b)` representing `a=*b`
`Store(a, b)` representing `*a=b`

The point-to analysis can now be done usinga Datalog program storing pairs of variables `a` and `b` in intentional relation `PointsTo` if `a` may point to `b`. The program contains the following four rules with the same head:

```
PointsTo(x, y) :- AddressOf(x, y).
PointsTo(x, y) :- Assign(x, z), PointsTo(z, y).
PointsTo(x, w) :- Load(x, z), PointsTo(z, y), PointsTo(y, w).
PointsTo(x, y) :- Store(z, w), PointsTo(z, x), PointsTo(w, y).
```

We pay attention to the load rule (the third rule). *Eval-rule* (line 2 of algorithm 2) for this rule will be translated into the following one RAM statement:

```
search 0$ ∈ Load do
    query {1$ ∈ PointsTo | 1$[0] == 0$[1]} do
        query {2$ ∈ PointsTo | 2$[0] == 1$[1]} do
            project (0$[0], 2$[1]) into ∆PointsTo
        end
    end
end;
```

Because there are two derived predicates in the body, *Eval-Rule-Incr* (line 11 of algorithm 2) for this rule will be translated into two RAM statements. This correspond to the union operation in the definition of *Eval-Rule-Incr*:

```
search 0$ ∈ Store do
    query {1$ ∈ ∆PointsTo | 1$[0] == 0$[0]} do
        query {2$ ∈ PointsTo | 2$[0] == 0$[1]} do
            if((1$[1], 2$[1]) ∉ PointsTo) then
                project (1$[1], 2$[1]) into ∆PointsTo'
            end
        end
    end
end;
search 0$ ∈ Store do
    query {1$ ∈ PointsTo | 1$[0] == 0$[0]} do
        query {2$ ∈ ∆PointsTo | 2$[0] == 0$[1]} do
            if((1$[1], 2$[1]) ∉ PointsTo) thens
                project (1$[1], 2$[1]) into ∆PointsTo'
            end
        end
    end
end;
```

3.2 Adding Parallelism

It is now clear that each statement translated from a rule can be executed in parallel. Consider exmple 3.1, the two *search* statements in the *Eval-Rule-Incr* for the load rule can be executed in parallel. In fact, the assign rule will be translated into one statement, the load rule and the store rule will both be translated into two statements. All five statements can be executed in parallel. We mentioned in the previous chapter that the there are two levels of parallelism in the semi-naïve evaluation algorithm. This nested parallelism is automatically spread out in the way the RAM program is constructed.

Chapter 4

B-Trees

Chapter 5

Evaluation

Chapter 6

Conclusion

▶conclude on the problem statement from the introduction◀

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1995.

[2] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends® in Programming Languages*, 2(1):1–69, 2015.

Appendix A

The Technical Details

►. . .◄