Improving Performance of the Datalog Engine in Flix

Zijun Yu, 202203581

Master's Thesis, Computer Science

June 2024

Advisor: Magnus Madsen

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

Datalog is a simple yet powerful declarative logic programming language. It allows programmers to solve problems by stating facts and rules about the problem domain rather than writing algorithmic instructions. Flix is a new functional, imperative, and logic programming language, featuring first-class support for Datalog, enabling programmers to write and compose Datalog programs directly in Flix.

This thesis focuses on optimizing the Datalog engine in Flix by introducing parallelism and implementing a concurrent B+ tree, specifically the B^{link} tree. Our aim was to enhance the performance of Datalog evaluation by replacing the current red-black tree with the B^{link} tree, which supports concurrent operations.

The evaluation shows significant performance improvements with the B^{link} tree engine, achieving up to 5x faster execution in some benchmarks and an average improvement of more than 2x. These results demonstrate the effectiveness of parallelism and advanced data structures in optimizing Datalog evaluation, laying a solid foundation for future improvements.

Acknowledgments

I would like to thank my advisor, Magnus Madsen, for his excellent guidance and feedback throughout this project and for being a great teacher in both this and a previous course. His advice on both theoretical and engineering aspects has been of great value.
I am also grateful to my family for their unwavering support in allowing me to study abroad. My two years of study and life in Aarhus have been simple, joyful, and fulfilling.

Zijun Yu,
Aarhus, June 2024.

Contents

Chapter 1

Introduction

Datalog is a simple yet powerful declarative logic programming language. Programs written in a logic programming language are collections 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 its main features is the first-class support for Datalog, allowing programmers to write and compose Datalog programs directly in Flix.

The Datalog engine in Flix is not fully optimized at this stage. It is implemented as a library using the semi-naïve evaluation, as most modern Datalog engines do. The engine has two parts: it first compiles the given Datalog program into an intermediate representation, RAM (Relational Algebra Machine), represented as Flix values, and then 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 evaluation, which includes three parts: (1) a benchmark suite is proposed for evaluating the performance of the engine, (2) a new parallel statement is added to RAM, and (3) a concurrent data structure is implemented to enable the interpretation of the parallel statement. Specifically, 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 evaluation 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 the semi-naïve algorithm is implemented in Flix, specifically how the RAM program is constructed and interpreted. We then explain how parallelism can be introduced. Lastly, we briefly show the benchmark suite. In Chapter 4, we first introduce B-trees and B+ trees and how a concurrent B+ tree is achieved. In Chapter 5, we evaluate the performance of the trees we implement and the engine using these trees.

Chapter 2

Datalog

2.1 Datalog Syntax

A Datalog program consists of *facts*, which are statements that are held to be true, and *rules*, which describe 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(v1, v2)`, meaning that there is an edge from vertex `v1` to vertex `v2` 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:

```
Path(x, y) :- Edge(x, y).
Path(x, y) :- Edge(x, z), Path(z, y).
```

Intuitively, the first rule above says that if there is an edge from vertex `x` to vertex `y`, then there is a path from `x` to `y`. The second rule says that if there is an edge from `x` to `z` and there exists a path from `z` to `y`, then there is a path from `x` to `y` as well.

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

$$A_0 \text{ :- } A_1, \ldots, A_n.$$

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 `Path` 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.

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

D is called the *Extensional Database* (EDB) and P is called the *Intensional Database* (IDB). We will refer to D as the database and P as the Datalog program. Thus, predicate symbols are partitioned into two disjoint sets: *base* (or EDB or *extensional*) 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 T, called the *immediate consequence operator*, which derives new facts starting from known facts, using the rules of the program. Namely, T_{P_D} takes as input a set of facts I, applies the rules, and returns a set of facts $T_{P_D}(I)$, called the *immediate consequences* of I w.r.t. P_D. T_{P_D} can be defined formally using relational algebra. We refer to section 3.3 in [2] for the precise definition. In short, the immediate consequence of a single rule is as simple as doing a cartesian product over the relations in the body of the rule joined by the same variables and then projecting the variables in the head of the rule. The immediate consequence of a set of rules is then the union of the immediate consequences of each rule.

We say that a set of facts 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 facts 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:

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

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

2.3 Datalog Evaluation

The fixpoint semantics immediately leads to an algorithm to evaluate Datalog programs, called *naïve evaluation*. To define the algorithm, we first define a function called *Eval-Rule* that takes as input a rule r and a set of facts I containing the relations of the predicates appearing in the body of the rule, and returns immediate consequences of I w.r.t. r.

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 relations `Edge` and `Path` yields the following:

$$Eval\text{-}Rule(r, Edge, Path) = \{Path(\text{"}a\text{"},\text{"}c\text{"}), Path(\text{"}b\text{"},\text{"}d\text{"}), Path(\text{"}a\text{"},\text{"}d\text{"})\}.$$

We then define another function called *Eval* that evaluates a set of rules whose predicates in the head are the same. Consider a program P and a database D. Let $R_1, \ldots, R_n$ be the base relations and $P_1, \ldots, P_m$ be the derived relations. For each derived predicate symbol p_i, we define $Eval(p_i, R_1, \ldots, R_n, P_1, \ldots, P_m)$ 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 $r_1, \ldots, r_n$, the derived predicate symbols are $p_1, \ldots, p_m$, and the base relations are $R_1, \ldots, R_n$, Algorithm 1 performs the evaluation of P over D.

One shortcoming of the naïve evaluation is that in each iteration, all tuples computed in the previous iterations are recomputed. For example, in Example 2.3, the tuples `Path("a", "c")` and `Path("b", "d")` 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

$$A_0 \text{ :- } A_1, \ldots, A_n.$$

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:

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

Algorithm 1 Naïve Evaluation

1: **for** $i \leftarrow 1$ to m **do**
2: $\quad P_i \leftarrow \emptyset$
3: **repeat**
4: $\quad$ **for** $i \leftarrow 1$ to m **do**
5: $\quad\quad P_i' \leftarrow P_i$
6: $\quad$ **for** $i \leftarrow 1$ to m **do**
7: $\quad\quad P_i \leftarrow \text{Eval}(p_i, R_1, \ldots, R_k, P_1', \ldots, P_m')$
8: **until** $P_i = P_i'$ for all $1 \leq i \leq m$
9: **return** $P_1, \ldots, P_m$

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

Algorithm 2 Semi-naïve Evaluation

1: **for** $i \leftarrow 1$ **to** m **do**
2: $\quad \Delta P_i \leftarrow \text{Eval-rule}(p_i, R_1, \ldots, R_k, \emptyset, \ldots, \emptyset)$
3: $\quad P_i \leftarrow \Delta P_i$
4: **repeat**
5: $\quad$ **for** $i \leftarrow 1$ **to** m **do**
6: $\quad\quad \Delta P_i' \leftarrow \Delta P_i$
7: $\quad$ **for** $i \leftarrow 1$ **to** m **do**
8: $\quad\quad \Delta P_i \leftarrow \text{Eval-rule-incr}(p_i, R_1, \ldots, R_k, P_1, \ldots, P_m, \Delta P_1', \ldots, \Delta P_m')$
9: $\quad\quad \Delta P_i \leftarrow \Delta P_i - P_i$
10: $\quad$ **for** $i \leftarrow 1$ **to** m **do**
11: $\quad\quad P_i \leftarrow P_i \cup \Delta P_i$
12: **until** $\Delta P_i = \emptyset$ **for all** $1 \leq i \leq m$
13: **return** $P_1, \ldots, P_m$

It is obvious that the semi-naïve evaluation is parallelizable. There are two levels of parallelism in Algorithm 2. First, each invocation of *Eval-Rule-Incr* can be done in parallel, that is, the body of the 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. The body of the loop from line 1 to line 4 can be done in parallel as well.

2.4 Datalog Extensions and Datalog in Flix

Notable extensions to Datalog include negation, aggregation, and functions.

The core fragment of Datalog cannot express non-monotone properties [3]. To achieve this

expressibility, we need to extend the core Datalog with *negation*.

Example 2.4. The following Datalog program computes the set of pairs (x, y) for which there is no path from x to y using negation:

```
Path(x, y) :- Edge(x, y).
Path(x, z) :- Path(x, y), Edge(y, z).
Unconnected(x, y) :- Vertex(x), Vertex(y), not Path(x, y).
```

Negations introduce non-monotonicity, which means we might lose the property that every Datalog program terminates and has a unique minimal solution. Almost all Datalog engines support negation through the use of `stratified negation`. Informally, programs are not allowed to recurse through negation. In stratified Datalog, a program P is viewed as a sequence of programs $P_1, \ldots, P_n$ where the IDB of P_i becomes the EDB of P_{i+1}. If a predicate A is computed using negation of another predicate B, then B must be computed in a previous stratum. In the above example, the first stratum consists of the first two rules (computing the IDB `Path`), and the second stratum consists of the third rule.

Aggregates are an important feature of query languages, e.g. SQL, that allow us to summarize a large amount of data into a single value. In Datalog, an aggregate rule has the following form:

$$R(x_1, \ldots, x_n, F(x_0)) \text{ :- } S(x_1, \ldots, x_n, x_0).$$

Here F an aggregate function, for instance `count`, `sum`, `min`, `max`, etc, and $S(x_1, \ldots, x_n, x_0)$ is some rule body mentioning at least all variables of the head.

Example 2.5. The following Datalog program computes the component representative of each vertex in a graph (the rule for computing `Connected` is omitted):

```
ComponentRep(x, min(y)) :- Connected(x, y).
```

In order to guarantee the existence of a minimal model of every Datalog program, several complex conditions must hold. One approach is to insist that the domains over which we are aggregating are complete lattices and that the program is in a sense monotonic [4]. Intuitively, this means that adding more elements to the multi-set being operated upon can only increase (or decrease) the value of the aggregate.

Flix adopts this approach to aggregation. In Flix, we call such rules *constraints on lattices*, as opposed to *constraints on relations*. To create such constraints, we have to define the lattice operations (the partial order, the least upper bound, and so on) as functions and associate them with a type. The aggregate, or latticenal value, is placed as the last atom in the head of the rule, and separated with other atoms by a semicolon, instead of a comma.

The addition of functions is also necessary for better expressivity (e.g., string transformations, list operations). Arithmetic operations can also be seen as functions. The guarantee of termination and finiteness is lost and becomes a responsibility of the programmer with the existence of functions. For example, this simple rule does not terminate:

```
A(x + 1) :- A(x).
```

In Flix, we can make sure that the program terminates by adding a guard in the body where the guard can be any boolean expression or a function application that returns a boolean:

```
A(x + 1) :- A(x), if x < 6.
```

We now demonstrate how Flix supports Datalog by an example.

Example 2.6. The following Flix program computes the graph of the connected components out of a graph of nodes:

```
def componentRep(g: Graph[node]): #{ ComponentRep(node, node) | r } with Order[node] =
    let nodes = inject g.nodes into Node;
    let edges = inject g.edges into Edge;
    let reachability = #{
        Reachable(n, n) :- Node(n).
        Reachable(n1, n2) :- Edge(n1, n2).
        Reachable(n1, n2) :- Edge(n2, n1).
        Reachable(n1, n2) :- Reachable(n1, m), Reachable(m, n2).
        ReachUp(n1) :- Reachable(n1, n2), if n1 < n2.
        ComponentRep(n, rep) :- Reachable(n, rep), not ReachUp(rep).
    };
    solve nodes, edges, reachability project ComponentRep

def connectGraph(g: Graph[node]): #{ Edge(Set[node], Set[node]) | r } with Order[node] =
    let missingEdges = #{
        Component(rep; Set#{n}) :- ComponentRep(n, rep).
        Edge(c1, c2) :- fix Component(_; c1), fix Component(_; c2), if c1 < c2.
    };
    solve componentRep(g), missingEdges project Edge

def main(): Unit \ IO =
    let graph = {
        nodes = Set.range(0, 8),
        edges = Set#{(0, 4), (0, 7), (2, 3), (1, 6), (5, 6)}
    };
    let connectedGraph = connectGraph(graph);
    let result = query connectedGraph select (c1, c2) from Edge(c1, c2);
    println("${result}")
```

The first thing to notice is the first-class support for Datalog constraints. In Flix, Datalog constraints, or Datalog programs, are first-class values. For example, the function `componentRep` returns a Datalog program containing the relation `ComponentRep`. The "`| r`" in the type means that it could contain more relations. This makes this returned Datalog program can be joined with other constraints, e.g. last line in `connectGraph`, and type check will succeed.

Secondly, in Flix, we can project any `Foldable` data structure containing the correct type of tuples into and from a relation. Here we project two set of tuples into the `Node` and the `Edge`

relations and the query result of the Datalog constraints, i.e. the variable `result`, is a vector of tuples of two sets of nodes, or, the vector of edges between every pair of components.

Negation is used in the function `componentRep` and aggregation is used in the function `connectGraph`. The `Set` type in the Flix standard library has implemented the lattice operations so it can be used directly as an aggregate. The least upper bound operation on sets is the union operation. Therefore, in this example, the relation `ComponentRep` contains these three facts `(1, 1) (5, 1) (6, 1)`, and we can deduce `Comment(1; {1, 5, 6})` from it. The `fix` keyword is used to explicitly introduce a stratification.

Flix has some other features for Datalog that we have not mentioned here, such as *functional predicates*. For more details, we refer to the Flix documentation.

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 the semi-naïve algorithm is implemented in Flix, specifically how the RAM program is constructed and interpreted. We then examine how parallelism can be introduced. Lastly, we discuss 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 `Path`. The initialization part of the semi-naïve evaluation algorithm (lines 1 to 4) will be translated into the following RAM program:

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

The invocation of *Eval-Rule* for the derived predicate `Path` 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* conditions is that the former is more efficient, as the tuples are stored in a sorted tree and the query is interpreted using range queries 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 (lines 5 to 16) will be translated into the following RAM program:

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

In this program, `Path` is the only derived predicate. Calling *Eval-Rule-Incr* for `Path` includes the 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 `Path` appears in one atom in the body, so the *search* statement is used to iterate over the *New* relation `ΔPath`. 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 performs a point-to analysis of variables in a C program [5]. 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 using a Datalog program storing pairs of variables `a` and `b` in the 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 single 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 ΔPointsTo
        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 corresponds to the union operation in the definition of *Eval-Rule-Incr*:

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

3.2 Parallelization

It is now clear that each statement translated from a rule can be executed in parallel. Consider Example 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 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.

The first thing to do is to add a new type of statement to RAM, called `Par` (Parallel), in contrast to the existing `Seq` (Sequential) statement:

```
pub enum RamStmt[v] {
    case Insert(RelOp[v])
    case Merge(RamSym[v], RamSym[v])
    case Assign(RamSym[v], RamSym[v])
    case Purge(RamSym[v])
    case Seq(Vector[RamStmt[v]])
    case Par(Vector[RamStmt[v]])
    case Until(Vector[BoolExp[v]], RamStmt[v])
    ...
}
```

One thing to note is that in the previous text we refer to `search` and `query` as statements, whereas in the actual implementation they are cases of the `RelOp` enum, along with `If` and `Project`. All of them belong to the category of the `Insert` statement.

We then change the compiler to generate as many `Par` statements as possible. The new RAM program will be of the following form in general (we use `||` to denote parallel statements and `;` to denote sequential statements):

```
search 0$ ∈ ... do
    ...
end ||
search 0$ ∈ ... do
    ...
end ||
... ;
merge Δ... into ... ||
merge Δ... into ... ||
...                 ;
Until ... do
    search 0$ ∈ ... do
        ...
    end ||
    search 0$ ∈ ... do
        ...
    end ||
    ... ;
    merge Δ...' into ... ||
    merge Δ...' into ... ||
    ...                  ;
    Δ... := Δ...' ||
    Δ... := Δ...' ||
    ...           ;
end
```

When interpreting the RAM program, we will fire up one thread for each statement in a `Par` statement. The problem is then how to perform concurrent operations on the relations. We have mentioned that the tuples of a relation are stored using trees. More specifically, the tuples are stored in a red-black tree, where the keys are the tuples themselves, and the values are either `Unit` or the corresponding lattice value if lattice is used in the relation.

3.3 Red-Black Trees

After the parallelization, there will be more than one thread trying to insert tuples into the same relation, for instance, in Example 3.1. However, red-black trees do not support concurrent insertions. One naïve solution is to include a mutex lock in the red-black tree and change all insertion methods to acquire the lock at the beginning and release the lock at the end. This solution works but is not efficient, for usually there is more than one rule for deriving a predicate in a realistic Datalog program, and the thread will be blocked by the lock. We will not demonstrate the evaluation of this solution in the evaluation section, but we mention that it sees 1.0x - 1.2x speedup on the realistic tests in our benchmark.

In this section, we give a brief introduction to red-black trees. More specifically, we will look at what range query is and how it has made Datalog evaluation more efficient.

A red-black tree is a specialized binary search tree data structure. Compared to other self-balancing binary search trees, the nodes in a red-black tree hold an extra bit called "color" representing "red" and "black," which is used when re-organizing the tree to ensure that it is always approximately balanced.

Binary search trees are used to store ordered data. In our case, the data is the tuples of a relation (serving as keys), possibly associated with a lattice value (serving as values). In a binary search tree, each node stores a key and a value, and the left child of a node has keys less than the node's key, and the right child has keys greater than the node's key. This sorted property allows for an efficient range query.

A range query is a query that retrieves data in a given range. In our case, the keys are tuples, and they are sorted first by the first element, then by the second element, and so on. As we have illustrated in Example 3.1, the `query` statement is always of the form `x[0]==a, x[1]==b, x[2]==c, ...`. When we are doing a range query, if at the current node, the first element of the key is less than `a`, or if the first element of the key is equal to `a` but the second element of the key is less than `b`, we can skip the left child of the current node, and so on. Likewise, we can skip the right child for some node. Only when the equality is met for all the elements in the query condition, we will visit both the left child and the right child.

3.4 The Benchmark

We have implemented a small benchmark suite that contains 4 files, 13 programs in total, to evaluate the performance of the Datalog engine in Flix.

- `Closure.flix` contains four realistic programs that compute either the transitive closure or something similar (e.g., strongly connected component) on graphs.
- `Pointer.flix` contains two programs that perform point-to analysis as shown in Example 3.1. The difference between the two programs is in the order of the variables in the rule. We will discuss this in Chapter 6.
- `Intersect.flix` and `Propagate.flix` contain four and three programs respectively that are rather artificial. For example, the third program in `Propagate.flix` is as follows:

```
pub def propaget3(): Int32 =
    let p = #{
        A(slowId(x) + 1) :- Z(x), A(x).
        B(slowId(x) + 1) :- Z(x), B(x).
        C(slowId(x) + 1) :- Z(x), C(x).
        ...
        Z(slowId(x) + 1) :- Z(x), if x < 6.
    };
    let f = #{
        Z(0).
        A(0).
        B(0).
        C(0).
        ...
    };
    query p, f select x from A(x)
    |> Vector.length
```

`SlowId` is an identity function that takes a long time to compute by calling a recursive Fibonacci function inside. In the semi-naïve evaluation, each rule will produce a new one-element tuple for each relation, stopping at 6. Because we only have one rule for each predicate, adding parallelism should significantly speed up the evaluation, no matter using the naïve mutex lock solution or using an advanced concurrent tree.

Chapter 4

B-Trees

We have discussed in previous sections that the tuples of a relation are stored using red-black trees and we want to change to a concurrent data structure to facilitate parallelism. The new data structure should act like a set and support efficient range queries.

The B-tree and its variants are widely used and studied in database systems. Many papers have described how to perform concurrent operations on B-trees. In this section, we will give a brief introduction to B-trees and B+ trees, and then we will introduce a new variant of B-trees called B^{link} trees that support concurrent reading and writing. In the next chapter, we will evaluate the performance of the Datalog engine using B+ trees and B^{link} trees.

4.1 B-Trees and B+ Trees

The B-tree is a multi-way search tree. An m-way search tree is a tree where each node has at most m children, hence storing at most $m-1$ keys. The keys in a node are sorted, and the keys in the children of a node are in the range defined by the keys of the node.

A B-tree of order m is a tree which satisfies the following properties:

- Every node has at most m children.
- Every node, except for the root and the leaves, has at least $\lceil m/2 \rceil$ children.
- The root node has at least two children unless it is a leaf.
- All leaves appear on the same level.
- A non-leaf node with k children contains $k-1$ keys.

In B-trees, every node stores a range of keys and their corresponding values. Each internal node's keys, including the root node, act as separation values which divide its subtrees, as shown in Figure 4.1 For example, if an internal node has 3 child nodes (or subtrees), then it must have 2 keys:

a_1 and a_2. All values in the leftmost subtree will be less than a_1, all values in the middle subtree will be between a_1 and a_2, and all values in the rightmost subtree will be greater than a_2.

The fourth property means that B-tree is a balanced tree. The second property ensures that each node is at least half full. This implies that two half-full nodes can be joined to make a legal node, and one full node can be split into two legal nodes (with an element pulled out and put into the parent). These properties make it possible to delete and insert new values into a B-tree and adjust the tree to preserve the B-tree properties.

A B+ tree is a variant of a B-tree where the internal nodes do not store any values, or pointers to records; thus, all values are stored in the leaf nodes. In addition, a leaf node may include a pointer to the next leaf node to speed sequential access. This makes range queries easy to implement. Since in a B+ tree, values are not stored in internal nodes, all keys appear twice in the tree: once in an internal node and once in a leaf node. This leads to a design choice to be made. In Figure 4.1, if we denote all the keys stored in a subtree at pointer P_i as V_i, then any $v \in V_i$ can either satisfy $K_{i-1} < v \leq K_i$ or $K_{i-1} \leq v < K_i$. We prefer the former because it makes the implementation of search consistent. When performing a search for some k inside a node, we return the index of the first key that is greater than or equal to k. This index will always be the correct index of the pointer array to follow.

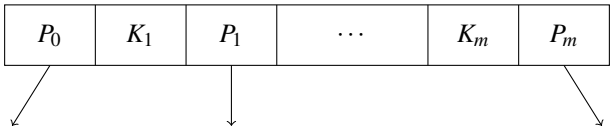

Figure 4.1: A B+ tree node

Compared to binary search trees, such as red-black trees, B-trees and their variants are more efficient in terms of I/O operations. B-trees are widely used in database systems to store indexes. The records in a database are stored on disk and pointers to the records are stored as values in the B-tree. The tree itself, serving as an index to the records, is also stored on disk. In our case, values are pointers to tuples in memory, and the tree is stored in memory as well. In binary search trees, there is a key and two pointers in each node, while in B-trees, m keys are paired with only $m+1$ pointers. This means two things. First, since a key is paired with fewer pointers, B-trees use less memory. Second, the tree is shallower than a binary search tree, so fewer pointers are followed to find a key. And since the keys are stored consecutively as an array in a node, the keys are more likely to be in the cache when searched inside a node. This means that B-trees are more cache-friendly than binary search trees.

Compared to B-trees, B+ trees have fewer pointers in the internal nodes. If we use a fixed size memory for a node, then B+ trees can hold more keys in one node, causing the tree to be shallower. And because we only store pointers to records in the leaf node, the structure of the leaf node is almost identical to the structure of the internal node. This also makes range query easier and more efficient since we only need to traverse the leaf nodes and the leaf nodes are linked together. The definition of B+ tree in Flix is as follows:

```
enum BPlusTree[k: Type, v: Type, r: Region](Region[r], Ref[Node[k, v, r], r])
enum Node[k: Type, v: Type, r: Region] {
  case Inner(Ref[Int32, r], Array[k, r], Array[Node[k, v, r], r])
  case  Leaf(Ref[Int32, r], Array[k, r], Array[v, r], Ref[Node[k, v, r], r])
  case Empty
}
```

Flix is a functional-first programming language that encourages the use of immutable data structures, but it has rich support for imperative programming with destructive updates to mutable data. It uses its effect system to separate pure and impure code. In particular, it uses the region concept to track the use of mutable memory. In Flix, all mutable memory belongs to a region that is tied to its lexical scope. When execution leaves the lexical scope of a region, all memory in that region becomes unreachable.

Flix has two basic types of mutable memory: References and Arrays. Higher-level mutable data structures, such as the B+ tree here, are built on top of Refs and Arrays.

A B+ tree contains a region, which tracks the lifetime of the tree, and a reference to the root node of the tree. A region is needed because other mutable data, such as the nodes of the tree, should have the same lifetime as the tree, so they need to be allocated using the same region as the tree. Therefore, the constructor will take a region as an argument, as well as the configuration of the tree, i.e., how many children can a node have at most. A new tree can then be created as follows:

```
def main(): Int32 =
    region r1 {
      let t = BPlusTree.empty(r1, 32);
      BPlusTree.size(t)
    }
```

A node in the B+ tree can be either an inner node or a leaf node. The `Empty` case serves as `Null` in other languages. The structure of the inner node is similar to that of the leaf node; they both contain an array of keys, an array of child nodes or values respectively, and an integer that stores the number of keys that currently exist in the node. The leaf node contains an extra pointer to the right sibling.

4.2 Insertion in B+ Trees

We now give the insertion algorithm for a B+ tree in Algorithm 3, which will be used in the evaluation in the next chapter and is also the base for the B^{link} tree. For convenience, the variable *keys* and *vals* are used to represent the key and value arrays in the node that is currently examined in the context.

The insertion algorithm is a recursive algorithm. It first descends to the leaf node where the key should be inserted. The descending function simply calls the search function to find the child node to follow and pushes the current node and the index of the child into a stack, and then repeatedly calls itself on the child node until it reaches a leaf node. The search function is a binary search

Algorithm 3 Insertion in B+ Tree

```
1:  function INSERT(t, k, v)
2:      root ← root(t)
3:      leaf, stack ← descendToLeaf(root, k)
4:      i ← searchNode(k, leaf)
5:      if i = num(keys) or k ≠ keys[i] then
6:          insertLeaf(k, v, i, stack, leaf, root)
7:      else
8:          keys[i] ← k
9:  function INSERTLEAF(k, v, i, node, stack, root)
10:     k', v' ← insert k and v into keys and vals
11:                                  ▷ k' and v' are the overflowed key and value if the array is full
12:     if k' and v' are null then
13:         increase the number of keys in the leaf by 1
14:     else
15:         newNode, upKey ← splitLeaf(node, k', v')
16:         put k' and v' into newNode
17:         if stack is empty then
18:             newRoot ← create a new inner node
19:             put upKey, leaf, and newNode in newRoot
20:             root ← newRoot
21:         else
22:             parent, i' ← pop(stack)
23:             insertInner(upKey, newNode, i', parent, stack, root)
24: function INSERTINNER(k, n, i, node, stack, root)
25:     same as insertLeaf, with the function of splitLeaf replaced by splitInner
```

function that returns the index of the first key that is greater than or equal to the key to be inserted, as we have discussed above.

The insertion function then checks if the key already exists in the leaf node, i.e. if the search reaches does not exceed the number of keys in the leaf node and the key is equal to the key at the index. If it exists, the value is updated. Otherwise, the key and value are inserted into the leaf node. The insertion may then cause the node to be split if the node is full. The split function will split the node into two nodes and return the key that should be inserted into the parent node. Recursive insertions are then performed using the stack. If we propagate all the way up until the stack is empty, then a new root node is created. There is one difference between splitting a leaf node and an inner node. As discussed, because values are only stored in leaf nodes in B+ trees, all keys appear twice in the tree. When splitting the leaf node, the middle key goes both into the new node and into the parent, while when splitting the inner node, the middle key leaves the current level and goes into the

parent. This satisfies the sorted property ($K_{i-1} < v \leq K_i$) of the B+ tree.

4.3 B^{link} Trees

There are many variants of B-trees designed to support concurrent reading and writing. In this project, we primarily studied the B^{link} tree proposed by Lehman and Yao [6]. We chose this work because it is implementable in Flix and meets our needs well. It focuses on concurrent reading and insertion, although deletion is poorly supported—which is acceptable for our purposes as we do not require deletion. Despite being an older work, it is a classic in the field, and some real-world programs, such as PostgreSQL, use B^{link} trees with certain improvements as their underlying data structure.

The idea of the B^{link} tree is to add another key-pointer pair to each inner node and an additional key to each leaf node. The structure of a node in a B^{link} tree is shown in Figure 4.2. The additional key is called the *high key*, i.e., the largest key in the subtree rooted at that node. For an inner node, the high key is the highest key in the subtree pointed by P_m. For a leaf node, the high key is the key is the same as K_m. The additional pointer is called *link* and it points to the right sibling. The node on every level is now linked together, hence the name B^{link} tree. Note this change still satisfies the sorted property of a node, that is, all keys in the subtree pointed by P_m are greater than K_m and less than or equal to the high key K_{m+1}.

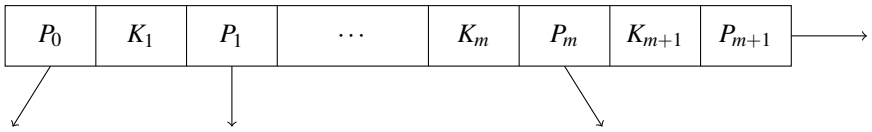

Figure 4.2: A B^{link} tree node

The B^{link} tree is essentially a fine-grained locking scheme. Each node is paired with a lock. When a thread is accessing the fields in a node, it has to acquire the lock first. The high key and the link then serve as a way to know if the current node is being split by another thread. When accessing a node, if the search key exceeds the high key, it means that the current node has been split and the thread should follow the link to the right sibling.

Figure 4.3 from Lehman and Yao shows an example of a B^{link} tree. Our specification differs from Lehman and Yao in that the rightmost node at each level has $+\infty$ as its high value, i.e., all the 99's in the diagram are changed to $+\infty$. Although the key $+\infty$ never appears in the tree, this decision is consistent as long as in the beginning when the empty root is created, $+\infty$ is set as the high value. We will see later that this will make the implementation easier in section 4.5.

The type definition of the B^{link} tree in Flix is as follows:

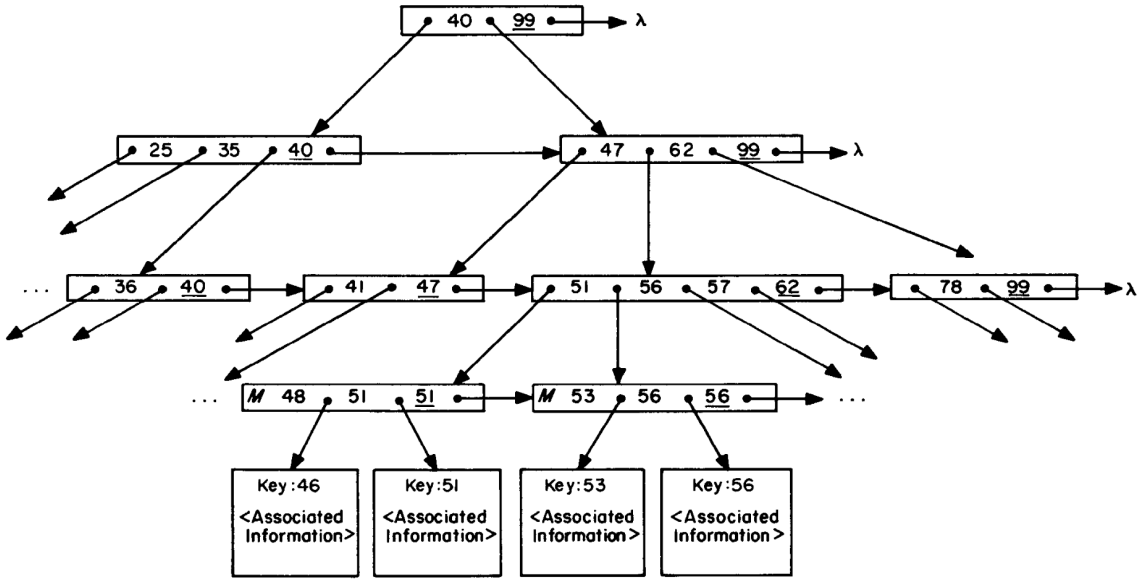

Figure 4.3: A B^{link} tree

```
enum BLinkTree[k: Type, v: Type, r: Region](Region[r], Ref[Node[k, v, r], r])
enum Node[k: Type, v: Type, r: Region] {
    case Inner(Int32, Ref[Int32, r], Array[k, r], Array[Node[k, v, r], r],
               Ref[Node[k, v, r], r], ReentrantReadWriteLock[r])
    case  Leaf(Int32, Ref[Int32, r], Array[k, r], Array[v, r],
               Ref[Node[k, v, r], r], ReentrantReadWriteLock[r])
    case Empty
}
```

The node structure is similar to that of the B+ tree, with the addition of the high key, stored in the last element of the key array, the link, and the lock. The original paper uses a simple mutex lock because in their setting, a node is stored as a block in the disk. Reading a block is an atomic operation so there is no locking involved in the `get` method. It is only when the intention of reading the block is to update it that the lock is acquired. In our setting, we need read-write locks because the nodes are stored in memory and when reading the fields in a node, we need to make sure that these fields are not being modified at the same time by another thread. The extra integer represents the level of the node in the tree. Leaf nodes are at level 0, and the level increases as we go up the tree. Once a node is created, the level is fixed and will not change. Level is needed here to deal with the case that Lehman and Yao did not discuss, that is, when the root node is concurrently split by other threads. We will discuss this in section 4.5.

4.4 Search in B^{link} Trees

To search for a value k in the tree, the search process begins at the root and proceeds by comparing k with the keys in each node in a path down the tree. At each node, we first acquire the read lock and then examine the keys and high key. If k is greater than the high key, we follow the link to the right sibling, unlock the current node, and repeat searching in the sibling. Otherwise, we examine the key array to find the index of the first key that is greater than or equal to k. Then we use the index to get the child node to follow. We unlock the current node, push it onto the stack, and repeat the search in the child node until we reach a leaf node.

The details of the search function are shown in Algorithm 4, following the same naming convention in Algorithm 3. Note how the high key is used to determine if we need to follow the link to the right sibling. In the descending function, even if we have found the correct child to follow at some level $i+1$, when we examine that child node at level i, it is still possible that it has been split by another thread, and the key we are looking for is moved to the right.

Locking is involved in the function `searchInner` and `searchLeaf`. The difference is that in `searchInner`, when we have finished searching the current level and have obtained the correct child node to follow, we release the lock as we will then proceed to the next level. In `searchLeaf`, when the function returns, the lock is still held, as the node needs to be examined further depending on if we are only reading or doing insertion.

4.5 Insertion in B^{link} Trees

To insert a key-value pair, we first perform operations similar to that for searching. Beginning at the root, we scan down the tree to the leaf node that should contain the key, keeping track of the rightmost node that we examined at each level in a stack.

The insertion of the key-value pair into the leaf node may necessitate splitting the node. In this case, we split the node as we have seen in the B+ tree, with the addition of changing the high key. The high key of the new node is set to the high key of the old node, and the high key of the old node is set to the last key that remains in the old node after splitting, which is then pushed up to the parent node recorded in the stack. The parent node, too, may need to be split. If so, we backtrack up the tree, splitting nodes and inserting new pointers into their parents. In all cases, we lock a node before modifying it. Just as we discussed in the insertion algorithm of the B+ tree, the splitting of the inner node and the leaf node slightly differs. When splitting the leaf node, the high key is pushed up to the parent, while when splitting the inner node, it is the middle key which neither remains in the old node nor is put in the new node that is pushed up to the parent.

The details of the insertion algorithm are shown in Algorithm 5, again following the same naming convention in Algorithm 3, where *keys*, *vals*, and *level* are used to represent the key array, value array, and the level of the node that is currently examined in the context, i.e., $node'$.

We mentioned before that unlike the original paper, for the rightmost node at each level, the high key is set to $+\infty$ instead of the actual largest key in this subtree. This makes splitting the rightmost

leaf node easier. When the rightmost leaf is split, we can always safely set the high key of the new node to that of the old node, which is $+\infty$, regardless of if we have inserted a key that is greater than all existing keys. This makes splitting the rightmost node consistent with splitting other nodes.

We also mentioned that Lehman and Yao fail to discuss what must happen when the root node becomes full and must be split. When an inserter recurses up the tree, it is possible it has reached the root, but the root level has been split by other threads, and a new root has been created. In this case, the stack made while descending does not help in finding the correct parents anymore. We adopt the solution from PostgreSQL. PostgreSQL has adopted B^{link} trees as the base for its index structure. Their solution is that when this happens, we re-descend the tree until we reach the level one above the old root [7]. Typically, the re-descending only needs to examine one level of nodes because, in most cases, if the old root level has been split, there will be enough space for insertion, and re-descending is not needed. It is rare for the root level to be split and quickly filled up again by other threads. In this case, re-descending is necessary, but it is highly unlikely that even the new root level is split and filled up.

This locking scheme is efficient in that only a constant number of nodes are locked at any time. The original paper couples locks as part of moving right when relocating a child node's downlink during an ascent of the tree. This is the only point where it has to simultaneously hold three locks (a lock on the child, the original parent, and the original parent's right sibling). However, in our implementation, an inserter never couples any locks, neither between the child and the parent, nor between siblings. This means that only one lock is held at any time. We will not provide a rigorous proof of the correctness here, but we argue that as long as we always obtain the correct pointer to follow from the current node when the current node is locked, and we always lock the node the pointer points to before accessing its fields, the implementation is correct.

4.6 Range Query

The implementation of range queries in B+ trees and B^{link} trees is similar. We find the first key that is greater than or equal to the left bound of the range in the inner node, with moving to the right involved in B^{link} trees, and then traverse down the tree to the leaf node. This is exactly the same as in searches and insertions. The `forAll` function simply follows the leftmost child in each node and traverses the leaf level all the way to the rightmost leaf node.

In B^{link} trees, we always lock the read lock when we are reading the fields in both inner nodes and leaf nodes. This enables threads doing queries or iterations to not block other threads from reading or writing the tree. Deadlocks could occur if the query or iteration tries to modify the tree. Programmers using these two functions should be aware of this and avoid modifying the tree in the query or iteration.

Algorithm 4 Search in B^{link} Tree

```
1:  function GET(k, t)
2:      root ← root(t)
3:      leaf, stack ← descendToLeaf(k, root, [])
4:      leaf′, i ← searchLeaf(k, leaf, false)
5:      if i = num(keys) or k ≠ keys[i] then
6:          release read lock of leaf′
7:          return None
8:      else
9:          v ← vals[i]
10:         release read lock of leaf′
11:         return Some(v)
12: function DESCENDTOLEAF(k, node, stack)
13:     if node is leaf then
14:         return node, stack
15:     else
16:         child, stack′ ← searchInner(k, node, stack)
17:         return descendToLeaf(k, child, stack′)
18: function SEARCHINNER(k, node, stack)
19:     acquire read lock of node
20:     if needToGoRight(k, node) then
21:         sibling ← link(node)
22:         release read lock of node
23:         return searchInner(k, sibling, stack)
24:     else
25:         i ← findFirstGreaterOrEqual(k, keys)
26:         child ← vals[i]
27:         release read lock of node
28:         return child, node :: stack
29: function SEARCHLEAF(k, node, isWriting)
30:     acquire read or write lock depending on isWriting
31:     if needToGoRight(k, node) then
32:         sibling ← link(node)
33:         release read or write lock depending on isWriting
34:         return searchInner(k, sibling, isWriting)
35:     else
36:         i ← findFirstGreaterOrEqual(k, keys)
37:         return node, i
38: function NEEDTOGORIGHT(k, node)
39:     return link(node) ≠ null ∧ k > high key(node)
```

Algorithm 5 Insert in B^{link} Tree

```
1: function PUT(k, v, t)
2:     root ← root(t)
3:     leaf, stack ← descendToLeaf(k, root, [])
4:     insertLeaf(k, v, leaf, stack, root)
5: function INSERTLEAF(k, v, node, stack, root)
6:     node', i ← searchLeaf(k, node', true)
7:     if i < num(keys) and k = keys[i] then
8:         vals[i] ← v
9:         release write lock of node'
10:    else
11:        k', v' ← insert k and v into the key and value array
12:        if k' and v' are null then
13:            increase the number of keys in the node' by 1
14:            release write lock of node'
15:        else
16:            newNode, upKey ← splitLeaf(node', k', v')
17:            release write lock of node'
18:            if stack is empty then
19:                if level(root) = level then
20:                    newRoot ← create a new inner node
21:                    put upKey, node', and newNode in newRoot
22:                    root ← newRoot
23:                else
24:                    redescend(upKey, newNode, level, root)
25:            else
26:                parent ← pop(stack)
27:                insertInner(upKey, newNode, parent, stack, root)
28: function INSERTINNER(k, n, node, stack, root)
29:    same as insertLeaf, with the function of splitLeaf replaced by splitInner
30: function REDESCEND(k, n, toLevel, root)
31:    node, stack ← descendToLevel(k, toLevel, root, [])
32:    insertInner(k, n, node, stack, root)
33: function DESCENDTOLEVEL(k, toLevel, node, stack)
34:    if level(node) = toLevel + 1 then
35:        return node, stack
36:    else
37:        child, stack' ← searchInner(k, node, stack)
38:        return descendToLevel(k, toLevel, child, stack')
```

Chapter 5

Evaluation

In this chapter, we address the following research questions:

RQ0 Does our implementation of the trees satisfy the asymptotic bound?

RQ1 Is insertion and range query in the B+ tree faster than in the built-in red-black trees?

RQ2 Is insertion in the B^{link} tree faster than in the B+ tree and red-black tree?

RQ3 Is Datalog evaluation using B+ trees faster than using red-black trees?

RQ4 Is Datalog evaluation using B^{link} trees faster than using B+ trees and red-black trees?

The experiments were conducted on a machine with 64 GB of memory and an Intel Core i5-13500 CPU with 14 threads.

To answer questions 0 and 1, we performed the following experiment: we created trees where both the keys and values were integers. The value was set to be the same as the key since it does not impact the experiment. We then populated the tree with a certain number of randomly generated keys. We measured the time taken to insert another key-value pair into the tree. Because the time complexity of insertion in both B+ trees and red-black trees is $O(\log_2 n)$ when binary search is used in B+ trees, dividing the time by $\log_2 n$ should yield a constant. To ensure accuracy, we inserted 100 random key-value pairs and divided the time by 100. Since the tree was initially populated to a size much larger than 100, dividing by $100 \log_2 n$ is reasonable even though n is changing.

Unless otherwise specified, the random numbers were generated with a fixed seed to ensure the same set of keys for different trees. We ran the experiment five times and took the average time for benchmarking the trees.

The results of the insertion experiment are shown in Figure 5.1, where the size of the trees ranged from 2^{17} to 2^{21}. We observed that both B+ trees and red-black trees satisfy the asymptotic bound, as evidenced by the nearly flat lines. Insertion in B+ trees is clearly faster than in red-black trees because, in B+ trees, the key array is more likely to be in the cache when comparing keys. The B+ tree with order 16 was slower, while there was no significant difference between B+ trees with

orders 32, 64, and 128. It is important to note that the built-in red-black trees in the Flix standard library are optimized and benchmarked, while our B+ trees were implemented quickly and simply.

B+ trees are often used in database systems where nodes are stored as blocks on a disk, so the size of a node is usually determined to fit within one block, minimizing I/O operations. In our case, nodes are stored in memory, and it is acceptable for a node not to fit within a cache line. Smaller nodes fit better in CPU caches, potentially leading to faster access times within each node, but it increase the tree height and traversal time. We expected that B+ trees with too small a node size would be slower because, since the L1 cache is usually large enough to hold a node, making the tree height a more important factor. The results met our expectations, with the B+ tree of order 16 being slower than the other three.

The range query results are shown in Figure 5.2. We first populated the tree with 2^{20} keys, shuffled randomly. We then measured the time of querying different ranges, where the left bound was half the size of the tree, i.e., 2^{19}, and the range sizes were 2^6, 2^{10}, 2^{14}, and 2^{18}. We expected that range queries on a small range would be faster in B+ trees than in red-black trees, which was confirmed by the experiment. However, as the range size increased, e.g., 2^{18} (one-fourth the size of the tree), B+ trees became slower than red-black trees. This is because, as explained in Section 4.6, the implementation of range queries in B+ trees requires many comparisons within each node to find the correct child to follow. Values are only stored in leaves, so we must traverse down to the leaves before performing any work. In contrast, in red-black trees, if the key falls within the range, we can immediately apply the given function to the value in the node during traversal.

To answer question 2, we populated the trees to a certain size with random integers and measured the time. For B^{link} trees, we used 14 threads (the number of cores in the machine) and evenly partitioned the input among the threads. The results in Figure 5.3 show that the best B^{link} tree is, on average, three times faster than the best B+ tree, which is again three times faster than the red-black tree. The B^{link} tree with order 32 performed best in the experiment, with order 64 also performing well. If nodes are too small, the tree height increases, leading to more frequent node splits, more locks, and longer times. Conversely, if nodes are too large, examining a node and inserting into it takes longer, and locks are more likely to compete. We also tested different combinations of the number of threads and the order of the tree and observed consistent results: using threads equal to the number of cores and orders of 32 or 64 performed best.

We did not test the range query for the B^{link} tree because its implementation is the same as for the B+ tree, with locks acquired if other threads modify the node, and no parallelism is available for a range query.

To answer questions 3 and 4, we ran the benchmark with four different implementations of the engine. The baseline is the built-in engine in Flix release 0.46.0, which uses red-black trees. Flix is a functional-first language that adopts the concept of regions to separate pure and impure code, as mentioned in Section 4.1. The definition of Datalog programs, including rules (constraints) and results (models), is as follows:

```
pub enum Datalog[v] {
    case Datalog(Vector[Constraint[v]], Vector[Constraint[v]])
    case Model(Map[RamSym[v], Map[Vector[v], v]])
```

```
        case Join(Datalog[v], Datalog[v])
    }
```

If we use a mutable data structure in the engine, we need to copy the results back into an immutable map after evaluating the entire Datalog program. We cannot simply change `Map` to `BLinkTree` because it would introduce a region variable `r` in the type, i.e., `Datalog[v, r]`. Therefore, for fair comparisons for the B+ tree and B^{link} tree, we added another engine where the only difference from the built-in engine is an extra copy of the results after evaluation. The results are shown in Figure 5.4.

Compared to the baseline, the B+ tree engine is, on average, slightly slower, partly due to the copying. When compared with the baseline with copying, the B+ tree engine is almost always faster, though only marginally. The B^{link} tree engine is significantly faster than both the baseline and the B+ tree engine. In the four realistic programs in `Closure.flix` and the first points-to analysis in `PointsTo.flix`, the B^{link} tree engine is 1.3x to 2.3x faster than the baseline, depending on the number of rules and their complexity. Generally, the more rules there are, the more we gain from parallelism. For example, in the `propagate3` program illustrated in Section 3.4, the B^{link} tree engine is 5 times faster than the baseline. For the rest of the programs, the B^{link} tree engine is on average 2.6 times faster than the baseline.

The only exception where both the B^{link} tree engine and the B+ tree engine are slower than the baseline is in the `points2` program. The difference between `points2` and `points1` is that two atoms in the body of the load rule are swapped, causing a range query statement to fall back to a search statement in the compiled RAM code. The reason why the B^{link} tree engine is slower than the baseline in this case is unclear.

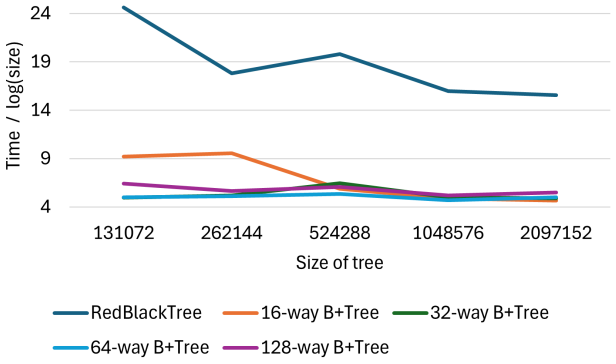

Figure 5.1: Time of inserting 1 value in Red-Black Tree and B+Trees

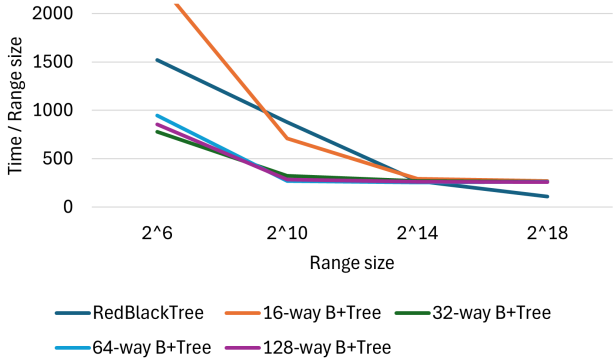

Figure 5.2: Time of range query in Red-Black Tree and B+Trees containing 2^{20} keys

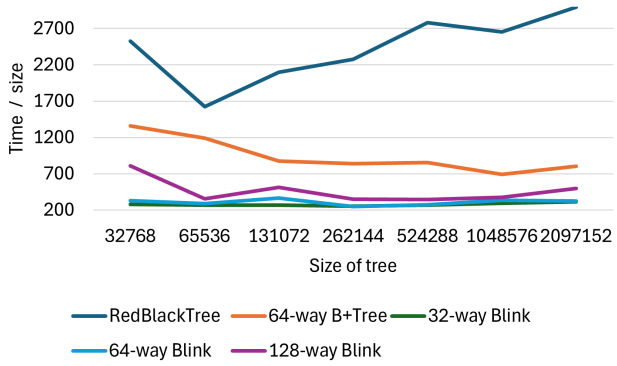

Figure 5.3: Time of populating diefferent trees

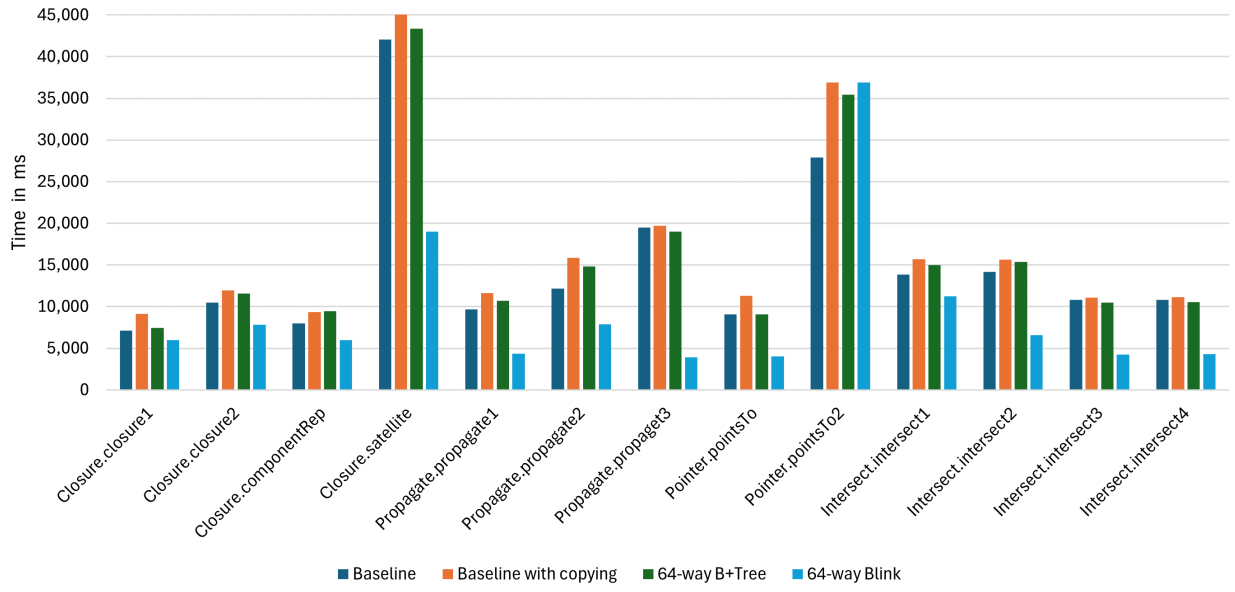

Figure 5.4: Benchmark of engines using different trees

Chapter 6

Future Work

6.1 Optimizing the B^{link} Tree

The current implementation of the B^{link} tree is a quick solution. There are numerous potential optimizations for both the algorithm and the code. For instance, there are alternative ways to implement the range query. The current approach finds the first key greater than or equal to the left bound at each level of the tree. This is not optimal because in the inner nodes, many comparisons of keys within the range will be revisited at the leaf level if the range is large. Since B^{link} trees have high keys, we could instead locate the node whose high key is the first to meet or exceed the left bound at each level. Only after reaching the correct leaf node would we examine the key array.

6.2 Index Selection and Rule Rewriting

We can also consider optimizations at the source level. For example, the benchmark `points1` contains the rules in Example 3.1. We have seen that the load rule will produce two range queries. However, programmers might not be aware of the underlying implementation of the engine and could write a load rule where the second and third atoms in the body are swapped. If so, the second atom would be compiled to a search statement as follows:

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

```
            query {2$ ∈ ΔPointsTo | 2$[0] == 0$[1] ∧ 2$[1] == 1$[0]} do
                project (0$[0], 1$[1]) into ΔPointsTo'
            end
        end
    end
end;
```

We can add an analysis phase in the compiler to reorder the atoms in the rules so that range queries are used as much as possible.

Additionally, we can analyze the program to determine the best index for each relation, i.e., the optimal order of the elements in the tuple when stored in a sorted tree data structure. For example, the benchmark `intersect3` contains rules similar to the following:

```
A(1 , i) :- Check(a, b, c), In(i, a, b, c).
A(2 , i) :- Check(a, _, c), In(i, a, _, c).
A(3 , i) :- Check(a, _, _), In(i, a, _, _).
A(4 , i) :- Check(a, b, _), In(i, a, b, _).
A(5 , i) :- Check(a, b, c), In(i, a, b, c).
```

where the EDB is `Check` and `In`, and the IDB is `A`. The current implementation compiles all five rules as search statements. However, if we analyze the program, we can actually store the tuples of the `In` relation as `(a, b, c, i)`. Then these rules can be translated into range queries.

Multiple indices can be created for the same relation. This means that one relation can have multiple trees storing the tuples in different orders of the elements. Although this can substantially improve performance, the memory footprint can become very large.

Index selection is studied and implemented by some other Datalog engines. For example, Souffle, a Datalog synthesis tool for static analysis, uses a strategy to automatically construct an appropriate set of indices [8].

6.3 Parallelism through Data Partitioning

Another direction for parallelism is to partition the tuples of the relations and let different threads evaluate the rules on different partitions. This parallelism is promising because the level of parallelism is not limited by the number of rules in the program or by skewed workload distribution. However, data partitioning is also challenging because we need a partitioning strategy that balances the workload of each thread and minimizes communication between threads. Communication means that threads may need to exchange their newly found facts in the IDB with other threads based on the partitioning strategy. A rich line of research on partitioning strategies for Datalog has existed since the 1990s. The monograph "Modern Datalog Engines" [9] lists a wealth of literature and Datalog engines on this topic.

6.4 Expanding the Benchmark Suite

Another future work is to enhance the benchmark suite to include more real-world Datalog programs and use actual input data instead of randomly generated ones. For example, in the field of program analysis, there are complex analyses with hundreds of relations and rules, and the input data extracted from real-world programs can be substantial. Developing a more comprehensive benchmark suite would provide a better evaluation of the Datalog engine's performance under realistic conditions.

Chapter 7

Conclusion

In this thesis, we explored the optimization of the Datalog engine in the Flix programming language, focusing on parallelism and efficient data structures. The primary goal was to improve the performance of Datalog evaluation by introducing parallelism and utilizing a concurrent B+ tree, specifically the B^{link} tree, to replace the red-black tree used in the current implementation.

We began by introducing Datalog, its syntax, semantics, semi-naïve evaluation, and the evaluation algorithm as implemented in Flix. The introduction of parallelism into the Datalog engine was then discussed, highlighting the addition of a new parallel statement to the compiled Datalog program and the implementation of the B^{link} tree as a concurrent data structure.

Our evaluation involved comparing the performance of the B^{link} tree engine against the baseline red-black tree engine. The results demonstrated that while the B+ tree engine showed marginal improvements over the baseline, the B^{link} tree engine provided significant performance gains, especially in programs with a higher number of rules. The B^{link} tree engine was up to 5x faster in some benchmarks, and on average more than 2x faster, illustrating the benefits of parallelism and efficient data structures in Datalog evaluation.

Throughout this project, I learned valuable lessons about the Flix programming language, Datalog, and the implementation of concurrent data structures. The region-based memory management in Flix was particularly interesting as a way to ensure manageable and safe use of mutable data structures and imperative code, although it sometimes brought inconvenience, as seen in the evaluation chapter. Additionally, the study of Datalog evaluation provided insights into the challenges and opportunities in implementing and optimizing Datalog engines, and laid a solid foundation for future research in this field.

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] Sergio Greco and Cristian Molinaro. Datalog and Logic Databases. Synthesis Lectures on Data Management, Cham, 2016. Springer International Publishing.

[3] F. Afrati, S.S. Cosmadakis, and M. Yannakakis. On datalog vs polynomial time. *Journal of Computer and System Sciences*, 51(2):177–196, 1995.

[4] Kenneth A Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. *Journal of Computer and System Sciences*, 54(1):79–97, 1997.

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

[6] Philip L. Lehman and s. Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, dec 1981.

[7] PostgreSQL. Readme for nbtree in postgresql source code, 2024. Available at: `https://github.com/postgres/postgres/blob/00ac25a/src/backend/access/nbtree/README#L112-L137`, commit hash: 00ac25a.

[8] Pavle Subotić, Herbert Jordan, Lijun Chang, Alan Fekete, and Bernhard Scholz. Automatic index selection for large-scale datalog computation. *Proc. VLDB Endow.*, 12(2):141–153, oct 2018.

[9] Bas Ketsman and Paraschos Koutris. Modern datalog engines. *Foundations and Trends® in Databases*, 12(1):1–68, 2022.

Appendix A

The Technical Details

The implementation of the Datalog engine using B+ tree and B^{link} tree can be found in the repository flix-datalog-engine and benchmark sutie can be found in the repository flix-datalog-benchmark.