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

Acknowledgments

►Write acknowledgments here. . . ◄

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 Datalog programs directly in Flix.

The Datalog engine in Flix is implemented 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: it compiles the given Datalog program into an intermediate representation, RAM (Relational Algebra Machine), represented as Flix values, and then it 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 **??**, 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 examine how parallelism can be introduced. Lastly, we discuss 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

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:

```
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 vertex `X` to 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 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.

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 called the *immediate 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 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 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 relations `edge` and `tc` yields the following:

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

We then define another function called *Eval* that 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 2.3 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 `tc("a", "c")` and `tc("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

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

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** $P_1, \ldots, P_m$

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:

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

We can now define the *semi-naïve evaluation* as shown in Algorithm 2.3.

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 2.3. 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. This is the main idea of the parallelization of this thesis. Also, the body of the 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 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 2.3 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 initialization part of the semi-naïve evaluation algorithm (lines 1 to 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* conditions is that the former is more efficient, as the tuples are stored using trees 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(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 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 `tc` appears in one atom in the body, so the *search* statement is used to iterate over the *New* relation `Δtc`. In Algorithm 2.3, 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 [3]. 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 2.3) 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 2.3) 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, as 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 later sections, but we mention that it sees 1.0x - 1.2x speedup on the realistic tests in our benchmark.

In this section, we will 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 a key less than the node's key, and the right child has a key 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 Benchmark

We have implemented a small benchmark suite that contains four 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. Indeed, when evaluating this program on a 14-core machine, we see a 6x speedup using the mutex lock solution.

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. A 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). This 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 pointers to records, thus all pointers to records 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 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 staisfy $K_{i-1} < v \leq K_i$ 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.

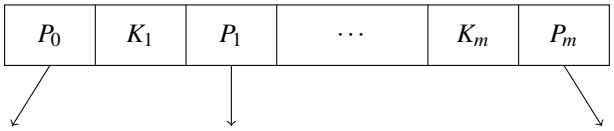

Figure 4.1: A B+ tree node

Compared to binary search trees, such as red-black trees, B-trees and its 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 in disk and pointers to the records are stored as values in the B-tree. The tree itself, serves as an index to the records, is also stored in 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 less pointers, B-trees use less memory. Second, the tree is shallower than a binary search tree so less 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 search inside a node. This means that B-trees is 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 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 constrcutor 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 searves 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 util 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 satisfy 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 that are designed to support concurrent reading and writing. In thisp roject, we mainly studied the B^{link} tree proposed by Lehman and Yao [2].

The idea of the B^{link} 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 satisfy 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}.

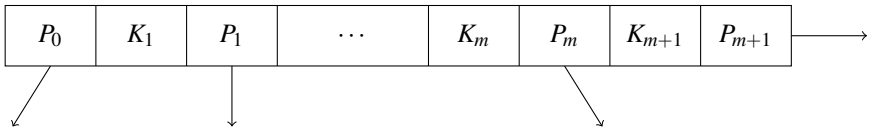

Figure 4.2: A B^{link} tree node

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 begning when the empty root is created, $+\infty$ is set as the high value. We will see later that this will the implementation easier in section 4.5.

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

```
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
}
```

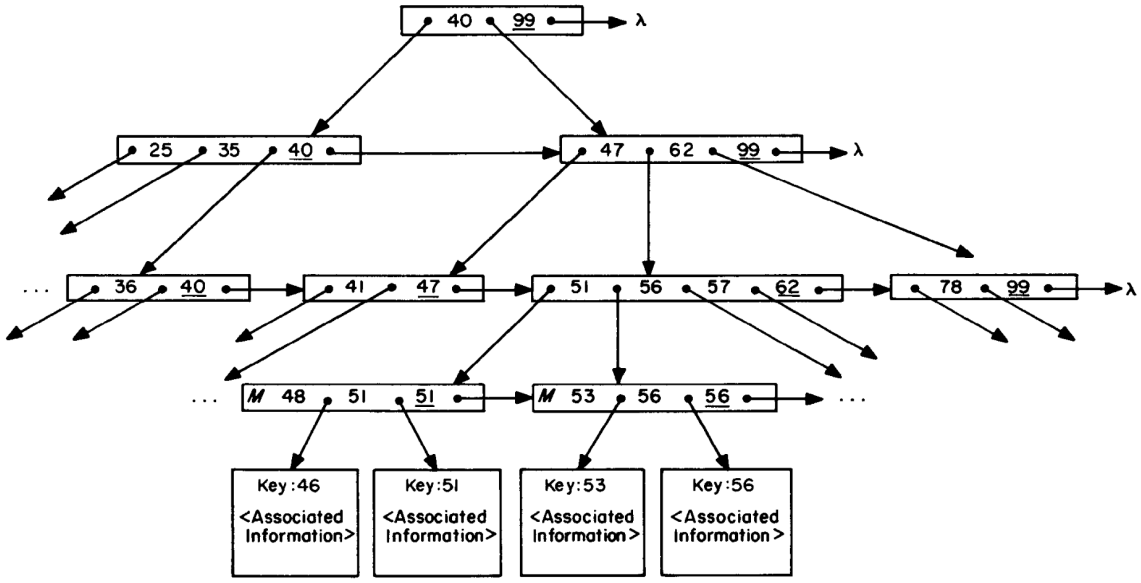

Figure 4.3: A B^{link} tree

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 then intension of reading the block is to update it that the lock is acquired. In our setting, we need to 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 fileds are not being modified at the same time by another thread. The extra ineteger represent 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 the stack, and repeat the search in the

child node until we reach a leaf node.

The details of the search function is shown in algorithm 4, following the same naming convention in 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 what 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 recoreded 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 what 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 be put in the new node that is pushed up to the parent.

The details of the insertion algorithm is shown in algorithm 5, again following the same naming convention in 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 spiltting 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 recursed up to the root but the root level has been split by other threads and a new root has been created. In this case the stack it made while descending does not help for finding the correct parents anymore. We adopt the solution from PostgreSQL. PostgreSQL has adpoted B^{link} trees as base for its index

structure. Their solution is that when this happens, we re-descending the tree until we reach the level one above the old root. 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 that the root level is split and is quickly filled up again by other threads. In this case, the re-descending is needed. 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 nevers 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 give 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.

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 answer the following five 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 map that use red-black trees?

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

To answer the question 0 and 1, we do the following experiment.

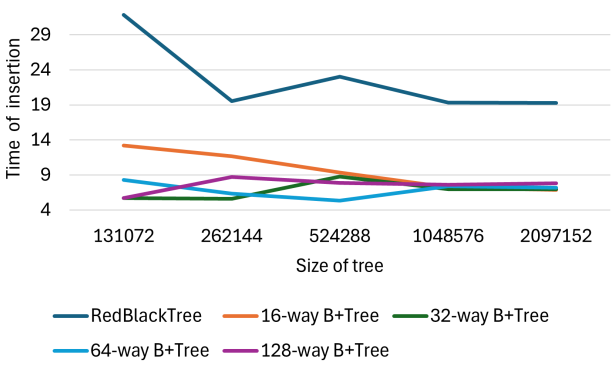

Figure 5.1: Time of inserting 1 value in Red-Black Tree and B+Trees

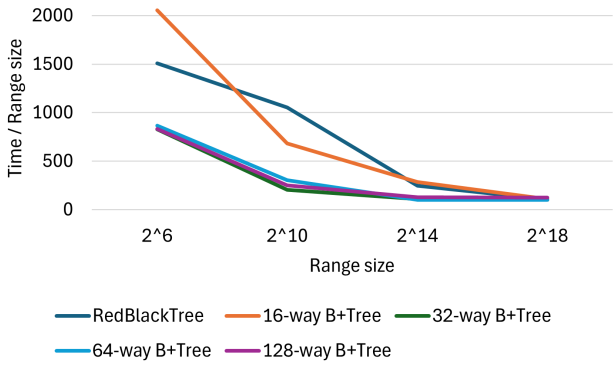

Figure 5.2: Time of range query in Red-Black Tree and B+Trees containing 2^{20} keys

Chapter 6

Future Work

Chapter 7

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] 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.

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

Appendix A

The Technical Details

►…◄