The Choice Construct in the Soufflé Language

Xiaowen Hu[1,2][0000−0002−4577−3360], Joshua Karp[1,3][0000−0002−4704−2182], David Zhao[1,4][0000−0002−3857−5016], Abdul Zreika[1,5][0000−0001−8812−5067], Xi Wu[1,6][0000−0001−5795−9798], and Bernhard Scholz[1,7][0000−0002−7672−7359]

[1] The University of Sydney, Australia
[2] xihu5895@uni.sydney.edu.au
[3] jkar4969@uni.sydney.edu.au
[4] dzha3983@uni.sydney.edu.au
[5] azre6702@uni.sydney.edu.au
[6] xi.wu@sydney.edu.au
[7] bernhard.scholz@sydney.edu.au

Abstract. Datalog has become a popular implementation language for solving large-scale, real world problems, including bug finders, network analysis tools, and disassemblers. These applications express complex behaviour with hundreds of relations and rules that often require a non-deterministic choice for tuples in relations to express worklist algorithms. This work is an experience report that describes the implementation of a *choice* construct in the Datalog engine Soufflé. With the choice construct we can express worklist algorithms such as spanning trees in a few lines of code. We highlight the differences between rule-based choice as described in prior work, and relation-based choice introduced by this work. We show that a choice construct enables certain worklist algorithms to be computed up to 10k× faster than having no choice construct.

Keywords: Static analysis, datalog, non-deterministic

1 Introduction

Datalog and other logic specification languages [25, 28, 4, 22] have become popular in recent years for implementing bug finders, static program analysis frameworks [25, 3], network analysis tools [39, 24], security analysis tools [31] and business applications [4]. For these applications, logic programming is used as a domain specific language to allow programmers to express complex program behavior succinctly, while enabling rapid-prototyping for scientific and industrial applications in a declarative fashion. For example, logic programming has gained traction in the area of program analysis due to its flexibility in building custom program analyzers [25], points-to analyses for Java programs [7], and security analysis for smart contracts [13, 12].

Although modern Datalog implementations such as Soufflé [34] have constructs (e.g., functors) that make Datalog Turing-equivalent, certain classes of algorithms are hard to implement. For example, worklist algorithms [33] that are commonly found in compilers and productivity tools [2], are challenging since

they require a non-deterministic choice from a set. Without the notion of choice, programmers must manually introduce an (arbitrary) ordering on a set and select the elements inductively to simulate this choice. The ordering and the inductive selection in Datalog requires dozens of rules and can be highly inefficient.

In database literature [27, 16, 8, 14, 15], there have been Datalog extensions for non-deterministic choice. In the work of Krishnamurthy, Naqvi, Greco and Zaniolo, the non-determinism is enforced operationally by introducing functional dependency constraints on relations. A functional dependency constraint enforces that a particular subset of values in each tuple (the key) can only occur once in the relation. For example, an ternary relation (x, y, z) with the functional dependency constraint $(x, y) \rightarrow z$ ensures that the two tuples $(1, 2, 3)$ and $(1, 2, 4)$ cannot simultaneously exist in the relation, since they both contain the same values $(1, 2)$ for the key (x, y). In this system, any tuple in the relation causes all subsequent tuples that violate the functional dependency constraint to be rejected from being inserted into the relation.

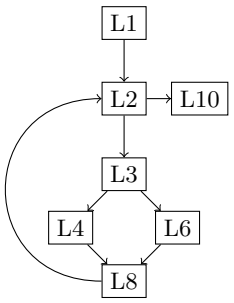
In this work, we report on the experience of implementing a choice construct in Soufflé [34, 25] and show (1) the simplicity of its semantics, (2) its ease of implementation, and (3) its efficiency in contrast to having no choice construct in the language. Prior work on choice has introduced functional dependencies as local, rule-based constraints, where the permissible tuples of a relation are only constrained on a rule-by-rule basis [16]. That work must be seen in the context of database research in the 90s that typically have a small number of rules. Soufflé programs have different characteristics, consisting of hundreds of rules and relations [7], where the relations are held in memory. For such applications, a rule-based choice becomes tedious and error prone because the functional dependency constraint may need to be repeated per rule. Hence, we introduce a new variant of choice called *relation-based choice*. A relation-based choice makes the underlying auxiliary relations of a ruled-based choice [10] explicit to the programmer. This approach is more amenable for logic programming with many relations/rules to ease the burden for the programmer.

The contributions of our paper are summarized as follows:

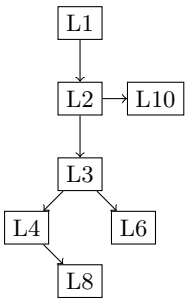
- We introduce a relation-based choice construct for the Soufflé (a Datalog engine) that enforces a global functional dependency upon a relation (not a rule). With a choice construct, algorithms such as worklists can be expressed effectively and efficiently.
- We show that the semantics of relation-based choice is easily implementable in an engine like Soufflé with its intermediate representation, called the Relational Algebra Machine (RAM).
- We explain the differences between the semantics of rule-based choice in prior work [10] and relation-based choice in Soufflé. We demonstrate that relation-based choice is easier to understand by users of large-scale Datalog programs.

```
1  x = 0;
2  while (x < 10) {
3    if (x % 2 == 0) {
4      print("fizz");
5    } else {
6      print("buzz");
7    }
8    x++;
9  }
10 return x;
```

(a) Example source code



(b) CFG of example



(c) Spanning tree of CFG

Fig. 1: Running Example, showcasing a snippet of source code with the corresponding control flow graph and spanning tree

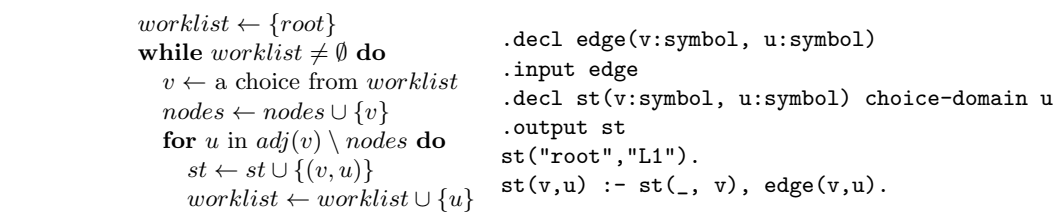
2 Motivating Example

Compilers and productivity tools require worklist algorithms [33], especially for control and data-flow analysis [2]. As part of more elaborate analyses, an example for a worklist algorithm is the construction of a spanning tree of a control-flow graph. This kind of application can be found for efficient placement of profiling code in programs [5], dataflow analysis [20, 35], and loop reductions [19].

Control flow graphs (CFGs) express the traversal of control in a program whose nodes are basic blocks (linear code) and edges of the graph indicate potential traversal between two basic blocks. Fig. 1a shows an input program whose control flow is depicted in Fig. 1b. The nodes in the control-flow graphs refer to the statements in the corresponding lines of Fig. 1a. The spanning tree of the CFG is illustrated in Fig. 1c, containing all the nodes of the CFG, but with only a subset of edges. Each node has at most one incoming edge and all nodes are connected, thus forming a spanning tree.

A standard worklist algorithm to compute a spanning tree is shown in Fig. 2a. A worklist contains all the nodes that ought to be visited in the next few iterations. The set `nodes` is used to store all visited nodes so far. The set `st` is used to store the edges of the spanning tree. The worklist is initialized with the root node, an artificial node with no incoming edge and a single out-going edge to the first basic block of the program. New nodes of the spanning tree are discovered and added to the worklist in each iteration, until no more valid nodes exist and the worklist becomes empty. Inside the loop, the worklist algorithm chooses an arbitrary node from the worklist. For this node, all adjacent nodes that haven't been visited yet will be added to the worklist and the spanning tree edges are constructed for the newly discovered nodes. With the worklist algorithm we can discover all reachable nodes and build the spanning tree in the discovery process.

While existing Datalog systems can be effectively used for many modern program analysis workloads [25, 7], worklist-style algorithms are often challenging.



(a) Worklist Algorithm (b) Soufflé with Choice

Fig. 2: Spanning Tree: Worklist Algorithm vs Soufflé with Relation-based Choice

Since standard modern Datalog engines are deterministic, they must explore *all* paths in a graph to compute a spanning tree, before making an arbitrary choice using a complex induction procedure. *Datalog* [1] represents programs as Horn clauses of the form $L_0 :- L_1, \ldots, L_n$. Each L_i has the form $R_i(x_1, \ldots, x_m)$; we say L_i is a *predicate* with relation R_i of arity m, and each attribute x_i is ether a constant or a variable. When the right hand side (the *body*) is empty, the Horn clause is interpreted as a fact; facts are unconditionally true. Otherwise, the Horn clause is interpreted as a rule, which means the head of the clause is true when all the literals in the body are evaluated to true: $L :- L_1, \ldots, L_n$. In particular, stratified negation [1], which is a standard semantics in Datalog to handle negation, does not permit a straightforward implementation of the worklist-style algorithms.

For example the spanning tree algorithm could be implemented with a rule such as `st(v,u) :- st(_,v), edge(v,u), !st(_,u)`. However, this is illegal in standard Datalog engines because it contains a negation that is not stratified [1], i.e., the recursive relation `st` depends on the negation of `st` itself. The *choice* construct for rules overcomes the problem of choosing elements [27], which also improves the overall expressive power of Datalog programs [15]. In this work, we introduce a variation of rule-based choice which we call a relation-based choice. Consider the spanning tree example expressed in the Soufflé language as illustrated in Fig. 2. The Datalog program imposes a functional dependency constraint for relation `st` with the keyword `choice-domain` on attribute `u`. The functional dependency constraint ensures that for a given value of attribute u there exists at most one tuple. For example, if the relation `st` already contains the tuple (`L5`, `L9`), a subsequent insertion of a tuple such as (`L7`, `L9`) whose `u`'s attribute value is `L9` will be suppressed. With that functional dependency, the relation `st` becomes a function whose domain is the attribute domain of v and its co-domain is the attribute domain of u. For sake of brevity, we omit the co-domain declaration in Soufflé so that all the excluded attributes of the domain specification implicitly become the attributes of the co-domain.

Without a choice construct, the notion of non-deterministic choice must be simulated via induction. This process is quite complex due to stratified negation. Stratification ensures that a simple expression of a complement set (i.e., to eliminate nodes that have already been visited) is impossible, since doing so

would involve a non-stratified negation. Instead, an algorithm written in stratified Datalog must construct an explicit complement relation, and use induction to select the next valid edge. Thus, while a spanning tree algorithm is expressible in modern Datalog engines (see Appendix of [23] for a Soufflé implementation), the native solution is very expensive in terms of runtime, memory usage, and code complexity.

To describe the native implementation in more detail, a rooted spanning tree is built incrementally from a chosen start node. The program repeatedly adds individual valid edges into the graph until no edges can be added. Since several edges may be valid at any given point, and we wish to explore only one arbitrary path, we must adorn the input edges with a total order so that ties among incoming edges can be broken. As the ordering is arbitrary, it is enough to assign a unique identifier to each edge in the graph. In Soufflé, unique numbers can be generated using the global counter, `$`, a unary functor which generates numbers sequentially when used, starting from the number zero (line 21). After creating an order among edges, an induction chooses the next valid edge from the worklist. A single valid edge must be chosen in each step, with elements with a lower ID being prioritized to break ties. We introduce a helper relation `chosenEdgeInductive` (line 127) with attribtues `step`, `edge_id` and `is_chosen` for constructing the induction. The `step` number identifies the current state of construction, incrementing with each new edge added into the spanning tree. For each step, we seed the induction with a dummy base case. The recursive rule then sequentially checks every edge, incrementing the edge ID being checked while they remain invalid. As soon as a valid edge is found, it is selected, and the recursive case terminates. A tuple in the relation contains a `TRUE` in the final column if and only if the edge with the given edge ID was chosen at that step. We cannot simply negate `validEdge` to check if an edge is invalid in the recursive rule for `chosenEdgeInductive`, since the validity of an edge relies on the choices made in previous steps, which in turn depends on this inductive rule again. Therefore, the assumptions of stratified negation would be broken. Instead, `invalidEdge` must be constructed positively alongside `validEdge`.

The resulting program requires deeply recursive rules using inductive arguments, the notion of total orders, and the positive construction of complement sets. Hence, the simulation of choice in logic is tedious and error-prone resulting in programs with sub-optimal performance. In contrast, the choice construct enables a much simpler and far more efficient expression of a spanning tree algorithm. In contrast to the 21 Datalog rules required for the native Soufflé implementation, the running example in Fig. 1c demonstrates an implementation with 1 rule and a choice constraint for the relation `st`.

3 Semantics of Choice

In the previous section, we established that a choice construct in a language like Soufflé is fundamental for implementing worklist style algorithms. However, there are two options for implementing choice in a Datalog engine. The choice

construct can be either (1) rule-based or (2) relation-based. In this section, we first explain the semantics of relation-based choice, which we choose to implement in Soufflé. We then explain the slight differences between the semantics of relation-based choice and rule-based choice. After that, we provide an example demonstrating why we believe relation-based choice makes more sense in modern Datalog language. Finally, we show that the expressive power of two different choice constructs are really the same and how to simulate rule-based choice semantics with relation-based choice construct.

Relation-based Choice. Relation-based choice extends the expressiveness of logic languages (e.g., Datalog) by introducing non-determinism into the logic framework at the relation level. In particular, choice constraints are declared for a relation, allowing programs to arbitrarily make a single choice out of a set of possible candidates. For example, a relation declared with choice constraints in the Souffle Language has the form:

$$\texttt{.decl}\ A(X_1, \ldots, X_n)\ \texttt{choice-domain}\ D_1, \ldots, D_k$$

Here, A is the relation name, and the sequence $X_1, \ldots, X_n$ forms the attributes of the relation. The choice constraints, `choice-domain` $D_1, \ldots, D_k$ imposes a set of relation-level constraints on the relation, where each domain D_i is a subset of attributes of the relation $D_1, \ldots, D_m \subseteq \{X_1, \ldots, X_n\}$. For example, a relation `A` declared with `.decl A(x:number, y:number, z:number) choice-domain x, (x,z).` has to respect two functional dependencies: $x \to (y, z)$ and $(x, z) \to y$. Semantically, each choice constraint D_i encodes a relation-level invariant which ensures that there is at most a single tuple in the relation for any particular value for the attributes in the choice domain. This constraint is similar to the notion of primary or candidate keys in a relational database [32].

We extend the standard fixpoint semantics of Datalog [1]. The choice construct must have the ability to arbitrarily *choose* tuples in a relation such that the resulting set of tuples satisfies the choice constraint. Consider a relation A with attributes $X_1, \ldots, X_n$. Let $D \subseteq \{X_1, \ldots, X_n\}$ be a choice domain, let M_A be the Cartesian product of the attribute domains of A, let $\mathcal{A} \subseteq M_A$ be a set of tuples for A, and let $\mathcal{A}\big|_D$ be the set of instantiated values when tuples in $\mathcal{A}$ are restricted to D. A *choice function* $c_D : 2^{M_A} \to 2^{M_A}$ on a set of tuples, $\mathcal{A}$, for the relation A can be defined as

$$c_D(\mathcal{A}) := \left\{\mathsf{SingleChoice}\left(\{t \in \mathcal{A} \mid t\big|_D \in \mathcal{A}\big|_D\}\right)\right\}$$

where $t\big|_D$ is the set of instantiated values for attributes in D for the tuple t. For each instantiation of attributes X_i in D, c_D chooses exactly one tuple matching that instantiation (via an extra function SingleChoice that arbitrarily chooses one element in the set). In other words, the choice function enforces uniqueness of values in the choice domain by arbitrarily choosing one tuple matching each instantiation. If M is the Cartesian product of the domain of relations in Datalog program P, then the choice function can be extended as $c : 2^M \to 2^M$, which applies c_D to each relation with choice constraints. The result of applying the

choice function c to a Datalog instance is an instance that satisfies the uniqueness condition of the choice constraints, by arbitrarily choosing one tuple for each instantiated set of values for each choice domain.

The other important semantics for choice constraints is to exclude tuples that already define values for the choice domain. The exclusion semantics applies for recursive rules, where an earlier iteration may define some values for the choice domain, while a later iteration computes the same values. In this situation, the tuples in the later iteration should be rejected, since those values in the choice domain are already chosen. Given another set of tuples $\mathcal{A}'$, the instantiations in D that are already defined in $\mathcal{A}$ can be excluded by the exclusion function:

$$e_D^{\mathcal{A}}(\mathcal{A}') := \mathcal{A}' \setminus \{t \in \mathcal{A}' \mid t\big|_D \in \mathcal{A}\big|_D\}$$

The exclusion function can also be extended to an instance I, where $e^I(I')$ applies exclusion for the whole instance, excluding tuples in I' where values for the choice domain are already defined in tuples in I.

We extend the standard semantics of Datalog with choice constraints such that the result of applying the consequence operator always satisfies these constraints (using bottom-up evaluation). For this, we define a *choice consequence operator*, Γ_P^c, which applies the exclusion and choice operations, to I as follows:

$$\Gamma_P^c(I) = I \cup c(e^I(\{t \mid t :- \ t_1, \ldots, t_k \text{ is a rule instantiation with each } t_i \in I\}))$$

It can be seen that $\Gamma_P^c(I)$ is monotone. Therefore, we can show that there exists a minimum fixpoint of $\Gamma_P^c(I)$ by using *Tarski's Fixpoint Theorem* [37]. The resulting fixpoint is denoted the *choice constraint model* of Datalog program P given instance I.

We extend the semi-naive evaluation (i.e., Algorithm Semi-Naive introduced in Appendix of [23]) with the choice consequence operator. The choice operator applies the choice and exclusion function and is similar to the consequence operator of semi-naive evaluation, defined as:

$$\Gamma_P^c(\Delta, I) = I \cup c\left(e^I\left(\left\{t \;\middle|\; \begin{array}{l} t :- \ t_1, \ldots, t_k \text{ with each } t_i \in I \\ \text{and at least one } t_j \in \Delta \end{array} \right\}\right)\right)$$

The Algorithm Semi-Naive in Appendix of [23] can then be modified by replacing the ordinary consequence operator Γ_P with the newly introduced choice consequence operator Γ_P^c. With this simple change, the efficient fixpoint evaluation of a choice program can be achieved.

Rule-based Choice. Unlike relation-based choice, rule-based choice from prior work enforces the functional dependency on the rule level. That is, only the tuples generated by the rules with the choice constructs have to respect the functional dependencies. Let's consider the rule-based choice version of the rooted spanning tree as an example.

```
st("root","L1").
st(v, u) :- st(_, v), edge(v,u), choice((u), (v)).
```

The keyword `choice((X), (Y))` specifies the functional dependency $X \rightarrow Y$ on the rule-level. Unlike the relation-based implementation, only the second rule in the above program has to respect the functional dependency, while the resulting relation `st` can still have a non-injecting relation between X and Y. In fact, the above program does not work as intended. Although the choice construct on second rule enforces that every end node u has a unique predecessor, there is nothing preventing the second rule from generating another edge to the starting node `L1`. This does not break the functional dependency because constraint is only enforced on rule-level and the tuple `st("root", "L1")` was specified in another clause in line one. To correct this, we need to rewrite the second rule as

```
st(v, u) :- st(_, v), edge(v,u), choice((u), (v)), u != "L1".
```

This program demonstrates a classic example where rule-based choice semantics can sometime become error-prone and hard to handle in large scale Datalog programs where each relation has dozens of rules.

Expressive Power. Although the user experience may differ, rule-based choice and relation-based choice have the same expressive power. We present an example of rewriting the rooted spanning tree example using rule-based choice semantics, but using relation-based choice construct. Consider the semantics of the rule-based choice implementation given under the stable model:

```
st("root","L1").
st(v, u) :- st(_, v), edge(v,u), chosen(u, v), u != "L1".
chosen(u, v) :- st(_, v), edge(v, u), !diffChoice(u, v).
diffChoice(u, v) :- chosen(u, v'), v != v'.
```

The above program cannot be computed under stratified semi-naive evaluation because of the cyclic negation between `chosen` and `diffChoice`. However, it is given by Giannotti et al.[10, 9] under the stable model to formally describe the semantics of the rule-based choice implementation. The intuitive meaning of the program is to use an auxiliary table (`diffChoice`) to record the generated tuples and prevent the rule from generating tuples that violate the dependency. The implementation given by Giannotti et al. follows this intuition, and uses an auxiliary table internally. To mimic the effect of this with relation-based choice, we use a separate relation `st'` with a relation-based choice constraint to act as the auxiliary table.

```
.decl st'(v:symbol, u:symbol) choice-domain(u)
.decl st(v:symbol, u:symbol)
st("root","L1").
st'(v, u) :- st(_, v), edge(v,u), u != "L1".
st(v, u) :- st'(v, u).
```

In Section 4 we show that because of how relation-based choice is implemented, this emulation does not suffer from any extra overhead and has the exact same cost as the one proposed in the literature where an auxiliary table is used.

4 Implementation in Soufflé

In the following, we describe the implementation of relation-based choice in the state-of-the-art Datalog engine Soufflé [25]. A general overview of the Soufflé infrastructure is shown in Fig. 3. Soufflé parses the input Datalog program into an Abstract Syntax Tree (AST) representation. After parsing, Souffle applies a series of high-level optimizations on the AST representation. The AST contains information including all declared relations, rules and facts of the source program. After applying the AST optimisations, the AST representation is lowered into an intermediate representation called the Relational Algebra Machine (RAM). A RAM program consists of a set of relational operations along with imperative constructs. Mid-level optimizations are then applied to the RAM code, which finally is synthesized into an equivalent C++ program (or is interpreted).

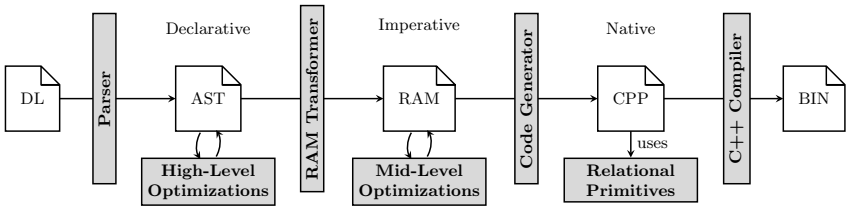


Fig. 3: Execution model of Soufflé.

A relation can be declared with zero or more choice constraints, each of which can contain a single attribute or a list of attributes. We extend the Soufflé parser to read a list of choice domains, written in the same form as shown in Section 3. We extend the current representation of relations in Souffle's with an extra attribute, storing each choice-domain as a list of indices representing the corresponding attributes' positions in the relation. For example, a relation declaration `.decl A(x, y) choice-domain x` will have a single choice-domain value $\{0\}$ denoting that the first attribute in A is in the choice-domain. A semantic check ensures that each choice-domain is valid (i.e., the attributes appear in the source relation), and a high-level optimization is used to reduce any redundant constraints.

At the final stage and before execution, we insert extra RAM operations to ensure the semantic for each insertion happens on a relation with choice-domain. We have various RAM elements implementing the semantics:

1. `TupleElement(t,i)` (or simply `t[i]`): It takes a runtime tuple $t = (t_1, \ldots, t_n)$ and an index i as arguments, and returns the value of the i^{th} element of t.
2. `Insert(t,R)`: It inserts a runtime tuple $t = (t_1, \ldots, t_n)$ into a relation R.
3. `ExistenceCheck(P,R)`: It checks if the given pattern $P = (p_0, \ldots, p_n)$ exists in the relation R. p_i can be either a runtime expression (e.g., `TupleElement`), a constant or a special value $\perp$ which matches with any value.

Input: AST representation of the source program.
Output: RAM representation with insertion guarded by existence check to guarantee the choice domain.
$RAM \leftarrow$ translate the AST into RAM without concerning choice
for each insertion `Insert(t, R)` in RAM **do**
  if R has choice-domain **then**
    $G \leftarrow$ a new `GuardedInsert(t, R, E=∅)`
    for each choice-domain C **do**
      add `ExistenceCheck(P, R)` into E, $p_i = t[i]$ if $i \in C$ else $p_i = \perp$
      if R has prefix `NEW` **then**
        $R' \leftarrow$ the corresponding original relation of R.
        add `ExistenceCheck(P, R')` into E.
      end if
    replace the existing insertion with G in RAM.
  end if
return RAM

Fig. 4: Augmenting a RAM program with Guarded Insertions

While inserting a tuple t into a relation R, the RAM program checks whether a choice constraints is violated. For this, we apply the choice function c and the exclusion function e^I mentioned in section 3. Before an `Insert(t,R)` operation (which would add tuple `t` to relation `R`), we add an extra check `ExistenceCheck(P,R)` with pattern $P = (p_0, \ldots, p_n)$. The value of p_i is defined as:

$$p_i = \begin{cases} \texttt{TupleElement(t, i)} & \text{if the } i^{th} \text{ attribute is in } D \\ \perp & \text{otherwise} \end{cases}$$

where D is the choice-domain $D = \{d_0, \ldots, d_k\}$ on R. If the existence check finds a matching tuple, then the insert operation is rejected. Thus, prior to the insertion tuples are filtered so that only tuples that do not violate the functional dependency constraint of the choice domain are inserted.

For a non-recursive rule, the relation R in the `ExistenceCheck` would be the original relation that the tuple is inserted into. However, for a recursive rule, the relation R would denote a *new* auxiliary relation rather than the original one (for semi-naïve evaluation), which requires the exclusion function. To achieve this in RAM, a similar existence check is applied to each version of the relation, i.e., if R has the form `R'`, then we also create an `ExistenceCheck(P,R')`, which ensures that any new tuples inserted into the relation will not replicate values for the choice-domains already defined in an earlier iteration, thus executing the semantics of the exclusion function.

To encapsulate the semantics of the filtering insertions, we introduce a new RAM operation, `GuardedInsert(t, R, E)`, i.e., a regular `Insert` operation with an extra field E representing a list of `ExistenceCheck` operations. The semantics of `GuardedInsert` specifies that the insertion only proceeds if all existence checks in E have been done. An algorithm is given in Fig. 4, demonstrating the process of translating a Soufflé program with choice constraints. In this algorithm each existing `Insert` operation is translated into a correspond-

```
INSERT ("root", "a") INTO new_st
READ INPUT INTO delta_st.
LOOP
  IF ((NOT (delta_st = ∅)) AND (NOT (graph = ∅)))
    FOR a IN delta_st
      FOR b IN graph ON INDEX b[0] = a[1]
        IF (NOT (⊥,b[1]) ∈ new_st) AND (NOT (⊥,b[1]) ∈ st)
          INSERT (a[1], b[1]) INTO new_st
   BREAK IF (new_st = ∅)
   MERGE new_st INTO st
   SWAP (delta_st, new_st)
   CLEAR new_st
END LOOP
```

Fig. 5: Resulting RAM program from spanning tree with relation-based choice

ing `GuardedInsert` operation, which encodes the semantics of the choice and exclusion functions.

With the new RAM transformation, the spanning tree program (Fig. 2b) is translated into the RAM program as shown in Fig. 5. The parts highlighted in blue are the extra existence checks introduced by the new translator. Because relation-based choice only requires extra existence checks , it is easy to see the emulation we describe in section 3 has the same cost as the rule-based choice implementation proposed in prior work.

Soufflé is equipped with highly-efficient data structures such as the specialized B-tree [26]. During the translation from RAM to C++, Soufflé analyzes the RAM representation to automatically compute indices for each primitive search [36]. This automatic index selection allows Soufflé to generate static C++ code that is tailored to data structures specialized for each index. As a result, the existence checks can be done efficiently with minimal overhead.

5 Experiments

This section explores the performance benefit of choice construct in Soufflé compared to native Soufflé without choice, as well as exploring any performance difference between relation-based choice and rule-based choice. Our experimental results illustrate that both choice constructs improve the environment of native Soufflé with similar performance statistics. Furthermore, we also demonstrate the applicability of choice and how it extends the expressive power of logic language. These experiments aim to answer three main research questions:

1. Does choice substantially improve runtime and memory performance over equivalent non-choice Datalog programs?
2. Does choice allow for easier expressivity for Datalog programs requiring nondeterminism?
3. Is there any performance difference between relation-based and rule-based choice?

Our experiments demonstrate a rooted spanning tree implementation applied on real-world input, along with 5 other algorithms that utilize choice constructs. For each algorithm, three versions are implemented:

1. **Relation-based Choice:** a Soufflé program that uses relation-based choice constraint (as implemented in Section 4)
2. **Rule-based Choice:** a Soufflé program that uses relation-based choice construct to emulate the rule-based choice semantics as described in Section 3.
3. **Native:** a Soufflé program that uses aggregates and auxiliary relations to emulate the effects of choice without using an explicit choice constraint

The experiments were conducted on a machine with an AMD Ryzen 2990WX 32-Core CPU and 126 GB of memory. All programs were run in sequential mode. Both runtime and memory usage were measured using the GNU `time` utility, observing both user time and maximum resident set size respectively.

5.1 Rooted spanning tree

We extract Control Flow Graphs (CFGs) from the real-world benchmark suite `SpecCPU2000` [21]. These CFGs consist of large graphs with small connected components, thus the spanning forest consists of one spanning tree for each connected component. Computing the spanning tree of a program's CFG is very important for program analysis tools to identify loops, possible optimization opportunities and security flaws, etc. Since each input file contains several connected components, we modify the rooted spanning tree example in Fig. 2b by computing a spanning forest with relation-based choice construct:

```
.decl edge(module:symbol, x:symbol, y:symbol)
.input edge
.decl startNode(module:symbol, x:symbol)
.input startNode
.decl st(module:symbol, x:symbol, y:symbol) choice-domain (module, y)
.output st

st(M,X,Y) :- startNode(M,X), edge(M,X,Y).
st(M,X,Y) :- st(M,_,X), edge(M,X,Y).
```

The attribute `module` identifies the name of the function where each connected component is generated from. By providing a single root node `startNode` for each component (line 4), we compute the spanning forest for the whole graph. The choice domain of relation `st` is specified as `(module, y)`, so that each module (connected component) contains a single spanning tree. Finally, the rule on line 9 states that a spanning tree edge from X to Y in the connected component M exists if the spanning tree reaches node X and there is an edge from X to Y.

The translated rooted spanning tree program in native Soufflé uses an inductive approach as in Section 2 and is modified in a similar way to calculate the spanning forest. Its implementation follows concepts from typical worklist algorithms, incrementally generating the set of edges corresponding to a spanning

	Benchmark Information		Runtime (seconds)		Memory usage (MBs)	
Program	# of components	average size (edges)	Native	Speedup factor	Choice	Native
gzip	84	28	2.75	275.00	5.00	10.95
swim	6	26	0.02	2.00	4.72	5.20
applu	16	56	1.42	142.00	4.84	8.69
gcc	1896	50	`timeout`	>10k	8.00	573.72
art	26	35	1.57	157.00	4.93	9.23
equake	26	16	0.22	22.00	4.87	6.11
ammp	175	32	26.19	2619.00	5.14	28.01
sixtrack	213	49	312.8	>10k	5.30	94.32
gap	830	38	298.2	>10k	5.84	116.64
bzip2	72	34	7.8	780.00	5.07	16.64
apsi	96	30	6.41	641.00	4.84	13.70
wupwise	20	32	1.7	170.00	5.02	9.94
mgrid	10	26	0.06	6.00	4.79	5.45
vpr	261	22	18.4	1840.00	5.18	21.84
mesa	1064	29	1258.55	>10k	5.98	237.61
mcf	26	25	0.26	26.00	4.82	6.36
crafty	108	88	1037.3	>10k	5.10	176.52
parser	293	25	54.79	5479.00	4.93	34.68
perlbmk	234	44	174.4	>10k	5.09	61.21
vortex	918	29	426.92	>10k	5.66	112.27
twolf	180	62	419.25	>10k	5.12	96.55

Table 1: Performance result from Spec CPU2000, timeout set to be 30 minutes.

tree of the input graph. The inductive process ensures that each edge appears only once in the output, and the output edges correspond to a tree, which contains no cycles.

During this experiment, we find no measurable runtime or memory difference between the relation-based and rule-based choice implementations. Both of them are able to finish all the benchmarks within 0.1 seconds and consume a similar amount of memory. Compared with relation-based choice, rule-based choice implementation requires an extra relation to keep track of the inserted tuples, and an extra insertion to dump the result from the auxiliary relation into the final result. However, in real-world use cases, because of the functional dependency constraint, the auxiliary relation tends to have a relatively small size, which makes the extra overhead small in comparison to the overall runtime and memory consumption. Specifically, in this experiment, the auxiliary relation in the rule-based choice version contains only the edges of the result spanning tree, which is much smaller than the overall graph size. Thus, we calculated a speed-up factor based on two choice implementations to demonstrate the performance difference between the choice constructs and native Soufflé implementation in Table 1.

		Relation-based Choice			Rule-based Choice			Native		
Program	Input	R#	T(s)	M(MB)	R#	T(s)	M(MB)	R#	T(s)	M(MB)
Eligible advisors	3000	1	0.01	5.5	2	0.01	5.7	4	0.11	13.7
Total order	2000	2	0.23	5.2	3	0.23	5.2	3	75.88	43.9
Bipartite matching	3000	1	2.73	93.2	2	2.73	93.2	15	`timeout`	771
More dogs than cats	18 000	3	4.42	7	4	4.42	7	1	0.01	6.7
Highest mark in grade	10 000	1	0.02	6	2	0.02	6.3	4	0.02	6.3

Table 2: Summary of experiment results.

The results show a significant improvement for the choice-based program compared to the native Soufflé program, performing at least 2× faster and up to more than 10k× faster on larger benchmarks such as `gcc` and `mesa`. In terms of memory consumption, the choice version consumes considerably less memory than the native Soufflé version, and achieves a consistent memory usage across all benchmarks. In comparison, the native Soufflé version uses significantly more memory as input size increases. This is because the choice constraint only computes and stores edges that are included in the spanning tree, which are generally fairly small compared to the constant overheads of executing a Soufflé program. On the other hand, the native version needs to store many intermediate computations and relies on a complex recursive scheme to obtain the same results.

Another consideration is the code complexity of both the choice constructs and native Soufflé implementation. For this spanning tree problem, the native Soufflé implementation requires 21 rules with complex recursive structure. On the other hand, relation-based choice version requires a minimum amount of code, with only 2 rules and a choice construct on the `st` relation. Finally, for rule-based choice, two extra auxiliary rules and one extra constraint are used as described in Section 3.

5.2 Other Applications

Along with the spanning tree example, we present five other algorithms, most of them are classic examples of non-deterministic algorithms in Datalog [9]:

- **Eligible advisors**: Choosing an advisor for each student.
- **Total order**: Assigning an arbitrary total order over an unordered list.
- **Bipartite matching**: Computing a matching over a bipartite graph.
- **More dogs than cats**: Taking two sets of elements and deciding if one set contains more elements than the other one.
- **Highest mark in grade**: Finding the highest mark in a subset of marks subject to a condition, e.g., the highest mark among students in each grade.

Table 2 shows the results for the choice versions compared to the native Souffé implementations. No runtime or memory difference is discovered between relation-based and rule-based choice. The reason is exactly the same as for the rooted spanning tree experiment, the overhead of rule-based choice implementation is extremely small because of the functional dependency constraint force

upon on the extra auxiliary relation. Thus, in the followings, we discuss only relation-based choice and native implementations, unless otherwise specified.

For the majority of these benchmarks, choice constraints lead to significantly better performance than the native Soufflé version. This improvement can be attributed to native Soufflé versions usually requiring the full computation of a relation, followed by selecting a unique subset satisfying the equivalent functional dependencies as a post-processing step. On the other hand, choice constraints allow for the functional dependencies to be checked on-the-fly, thus not needing the full unconstrained relation, benefiting both memory and runtime.

The *eligible advisors* example most clearly demonstrates the improvement in performance with the choice construct. Here, the relation-based choice can simply compute the student/advisor relationship with a single rule with a choice constraint on the `advisor` relation. However, the native Soufflé implementation must compute the full unconstrained `advisor` relation, with a unique numbering scheme to enforce a total ordering. Then, as a post-processing step, the algorithm selects a subset satisfying the choice constraint by using the total ordering (for example, by choosing the minimum value for the unique number).

Similar patterns can also be observed in the *total order* and *bipartite matching* examples. These benchmarks demonstrate situations where choice constraints allow for both an easier and more effective specification of the problem.

On the other hand, the benchmark *highest mark* shows a negligible performance difference. In both implementations, an aggregation is used to summarize the highest mark of each grade and is the main performance bottleneck of the whole algorithm. The performance benefit of the choice constraint that is used to restrict the result of the aggregation becomes insignificant. However, the difference in number of rules (4 v.s. 1) still demonstrate the expressiveness of the choice constraint.

The only benchmark where the native Soufflé implementation outperformes the choice version is *more dogs than cats*. In this example, the choice version consider building an injective function between the two set of elements, and then check if the domain covers all the codomain, if so, the size of the domain set is greater than or equal to the codomain set. On the other hand, the native implementation takes a more straightforward approach, using a simple count aggregate to compute the sizes of the relations.

Importantly, for all examples, the choice version uses equal or less memory compared to the native Soufflé counterpart. This improvement is a result of the auxiliary relations each native Soufflé program utilizes to perform their computations. The difference is most evident in the *total order* example, where the native Soufflé implementation suffers an approximate 850% increase in memory usage as a result of its auxiliary relations.

Going beyond performance results, every example is implemented more elegantly using choice constraints. For most of the benchmarks, the choice version contains less than half the number of rules of the native Soufflé version, and in three of the five benchmarks, the choice version contains only a single rule. While not a perfect measurement of elegance, the small number of rules indicates

that the choice-based implementations are generally more succinct and easier to understand than the native Soufflé versions. As shown, native Soufflé implementations of programs requiring arbitrary choice, as in worklist algorithms, typically involve the construction of several intertwined recursive relations with their complements, in addition to inductive rules, aggregate functions, and imposed total orderings. Such substantial overhead often obscures the meaning of the program. With the choice construct, such behavior is modeled with a simple constraint declaration. Moreover, the clearer semantics of the choice versions allows for a simpler extension and modification of the underlying program. For example, modifying the spanning tree example in Section 5.1 to constrain over only the attribute `y` rather than the pair `(module, y)` would involve changing only the given choice constraint. In a native Soufflé implementation, changing these functional dependencies could involve substantial structural changes to the auxiliary relations to ensure correctness.

In the context of Soufflé, these experiments demonstrate a significant impact of choice constraints, both in terms of performance overhead as well as the ease in expressing these algorithms. Thus, the introduction of choice constraints can be seen as extending the effective expressive power of the language, since certain problems that were infeasible using aggregates and auxiliary relations can now be solved using choice constraints.

6 Related Work

In relational databases, the notion of functional dependencies [38, 6] is an important concept that allows a database designer to encode certain uniqueness properties as an invariant on a relation. These invariants are enforced when the relation is modified, with the database system rejecting any data that violates the uniqueness constraint. In logic programming, a deterministic computation is expressed as a set of logic rules. To extend the capabilities of this framework, previous work has introduced the choice construct [27, 30] as a means of supporting non-determinism in Datalog, by enforcing uniqueness constraints similar to functional dependencies. There is some prior work on choice for Prolog [29]. Over the years, the applicability of choice has extended into the expression of greedy algorithms [16–18], as well as improving the overall expressive power of Datalog queries [11, 14, 15]. It has been cited to be particularly powerful when defining aggregate functions for relations, especially when used in conjunction with other predicates [8].

Choice constructs in prior work provide an intuitive foundation for enforcing non-determinism using a rule-based choice constraint, which is applied to a singular rule in the program, so that the underlying functional dependency is exclusively enforced on the local level of the specific rule that the constraint is declared on. In order to enforce these rule-based dependencies, auxiliary relations (e.g., the *chosen* relations in [16]) are required to provide an intermediate platform for computation for each rule with a constraint. The semantics of rule-

based choice can be tedious and error prone when applying on Soufflé's programs that consist of hundreds of rules and relations.

7 Conclusion

Extending the expressive power of logic languages is a pertinent research area, especially with these languages becoming increasingly used in real-world problems. While languages such as Datalog have found success in a number of areas, worklist-style algorithms require notions of non-determinism which is currently challenging in modern Datalog engines. In this work, we report on implementing a choice construct in the Soufflé Language. We experiment with two flavors of the choice construct: rule-based choice (that has been reported in prior work) and relation-based choice, which we introduce in this work.

We experiment with a number of classic algorithms using the two choice constructs and show that using a choice construct significantly improves the performance, along with greater elegance in expressing non-determinism in Datalog. Our experiments indicate that there is a negligible performance difference between the two flavors of choice constructs. However, we show with an example that the semantics of rule-based choice can be tedious and error prone in Datalog programs with a large number of rules and relations.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases: The Logical Level. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edn. (1995)
2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc. (2006)
3. Allen, N., Scholz, B., Krishnan, P.: Staged Points-to Analysis for Large Code Bases, pp. 131–150. Springer Berlin Heidelberg (2015)
4. Aref, M., ten Cate, B., Green, T.J., Kimelfeld, B., Olteanu, D., Pasalic, E., Veldhuizen, T.L., Washburn, G.: Design and Implementation of the LogicBlox System. pp. 1371–1382. SIGMOD '15, ACM (2015)
5. Ball, T., Larus, J.R.: Efficient Path Profiling. In: Proc. 29th Annual ACM/IEEE International Symposium on Microarchitecture. p. 46–57. MICRO 29 (1996)
6. Beeri, C., Fagin, R., Howard, J.H.: A complete axiomatization for functional and multivalued dependencies in database relations. In: Proceedings of the 1977 ACM SIGMOD international conference on Management of data. pp. 47–61 (1977)
7. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: Proc. 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications. pp. 243–262 (2009)
8. Ceri, S., Gottlob, G., Tanca, L.: Overview of Research Prototypes for Integrating Relational Databases and Logic Programming, pp. 246–266. Springer (1990)
9. Giannotti, F., Greco, S., Saccá, D., Zaniolo, C.: Programming with nondeterminism in deductive databases. Annals of Mathematics and Artificial Intelligence **19**, 97–125 (2004)

10. Giannotti, F., Pedreschi, D., Saccà, D., Zaniolo, C.: Non-determinism in deductive databases. In: Proc. Deductive and Object-Oriented Databases. pp. 129–146. Springer Berlin Heidelberg (1991)
11. Giannotti, F., Pedreschi, D., Zaniolo, C.: Semantics and expressive power of non-deterministic constructs in deductive databases. Journal of Computer and System Sciences **62**(1), 15 – 42 (2001)
12. Grech, N., Brent, L., Scholz, B., Smaragdakis, Y.: Gigahorse: Thorough, Declarative Decompilation of Smart Contracts. In: ICSE 19. pp. 1176–1186. ACM (2019)
13. Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y.: Madmax: Surviving out-of-gas conditions in ethereum smart contracts. In: SPLASH 2018 OOPSLA (2018)
14. Greco, S., Molinaro, C.: Datalog and Logic Databases. Synthesis Lectures on Data Management **10**, 47–57 (10 2016)
15. Greco, S., Saccá, D., Zaniolo, C.: DATALOG Queries with Stratified Negation and Choice: from P to D^P. In: ICDT (1995)
16. Greco, S., Zaniolo, C.: Greedy Algorithms in Datalog with Choice and Negation. In: IJCSLP (1998)
17. Greco, S., Zaniolo, C.: Greedy algorithms in datalog. Theory Pract. Log. Program. **1**(4), 381–407 (2001)
18. Greco, S., Zaniolo, C., Ganguly, S.: Greedy by choice. In: Proc. 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 2-4, 1992, San Diego, California, USA. pp. 105–113. ACM Press (1992)
19. Hecht, M.S., Ullman, J.D.: Characterizations of reducible flow graphs. Journal of the ACM (JACM) **21**(3), 367–375 (1974)
20. Hecht, M.S., Ullman, J.D.: A simple algorithm for global data flow analysis problems. SIAM Journal on Computing **4**(4), 519–532 (1975)
21. Henning, J.L.: SPEC CPU2000: Measuring CPU Performance in the New Millennium. Computer **33**(7), 28–35 (Jul 2000)
22. Hoder, K., Bjørner, N., de Moura, L.: μZ– An Efficient Engine for Fixed Points with Constraints. In: Computer Aided Verification. pp. 457–462. Springer (2011)
23. Hu, X., Karp, J., Zhao, D., Zreika, A., Wu, X., Scholz, B.: The choice construct in the souffle language (2021)
24. Huang, S.S., Green, T.J., Loo, B.T.: Datalog and Emerging Applications: An Interactive Tutorial. pp. 1213–1216. SIGMOD '11, ACM (2011)
25. Jordan, H., Scholz, B., Subotic, P.: Soufflé: On Synthesis of Program Analyzers. In: CAV 2016 Part II. LNCS, vol. 9780, pp. 422–430. Springer (2016)
26. Jordan, H., Subotić, P., Zhao, D., Scholz, B.: A Specialized B-Tree for Concurrent Datalog Evaluation. p. 327–339. PPoPP '19, ACM (2019)
27. Krishnamurthy, R., Naqvi, S.: Non-Deterministic Choice in Datalog. In: Proc. International Conference on Data and Knowledge Bases, pp. 416 – 424. Morgan Kaufmann (1988)
28. Madsen, M., Yee, M.H., Lhoták, O.: From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. pp. 194–208. PLDI '16, ACM (2016)
29. Mendelzon, A.O.: Functional dependencies in logic programs. In: VLDB - Volume 11. p. 324–330. VLDB Endowment (1985)
30. Naqvi, S.A., Tsur, S.: A Logical Language for Data and Knowledge Bases. Computer Science Press (1989)
31. Ou, X., Govindavajhala, S., Appel, A.W.: MulVAL: A Logic-based Network Security Analyzer. In: Proc. USENIX Security Symposium - Volume 14. pp. 8–8. SSYM'05, USENIX Association (2005)

32. Paredaens, J., De Bra, P., Gyssens, M., Van Gucht, D.: Constraints, pp. 61–112. Springer Berlin Heidelberg (1989)
33. Rayside, D., Kontogiannis, K.: A generic worklist algorithm for graph reachability problems in program analysis. In: Proceedings of the Sixth European Conference on Software Maintenance and Reengineering. pp. 67–76 (2002)
34. Scholz, B., Jordan, H., Subotić, P., Westmann, T.: On Fast Large-Scale Program Analysis in Datalog. p. 196–206. CC 2016, ACM (2016)
35. Sharir, M.: A strong-connectivity algorithm and its applications in data flow analysis. Computers & Mathematics with Applications **7**(1), 67–72 (1981)
36. Subotic, P., Jordan, H., Chang, L., Fekete, A.D., Scholz, B.: Automatic Index Selection for Large-Scale Datalog Computation. PVLDB **12**(2), 141–153 (2018)
37. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math. **5**(2), 285–309 (1955), https://projecteuclid.org:443/euclid.pjm/1103044538
38. Wiederhold, G.: Database design, vol. 1077. McGraw-Hill New York (1983)
39. Zhou, W., Sherr, M., Tao, T., Li, X., Loo, B.T., Mao, Y.: Efficient querying and maintenance of network provenance at internet-scale. SIGMOD pp. 615–626 (2010)