Flan: An Expressive and Efficient Datalog Compiler for Program Analysis

SUPUN ABEYSINGHE, ANXHELO XHEBRAJ, and TIARK ROMPF, Purdue University, USA

Datalog has gained prominence in program analysis due to its expressiveness and ease of use. Its generic fixpoint resolution algorithm over relational domains simplifies the expression of many complex analyses. The performance and scalability issues of early Datalog approaches have been addressed by tools such as Soufflé through specialized code generation. Still, while pure Datalog is expressive enough to support a wide range of analyses, there is a growing need for extensions to accommodate increasingly complex analyses. This has led to the development of various extensions, such as Flix, Datafun, and Formulog, which enhance Datalog with features like arbitrary lattices and SMT constraints.

Most of these extensions recognize the need for full interoperability between Datalog and a full-fledged programming language, a functionality that high-performance systems like Soufflé lack. Specifically, in most cases, they construct languages from scratch with first-class Datalog support, allowing greater flexibility. However, this flexibility often comes at the cost of performance due to the conflicting requirements of prioritizing modularity and abstraction over efficiency. Consequently, achieving both flexibility and compilation to highly-performant specialized code poses a significant challenge.

In this work, we reconcile the competing demands of expressiveness and performance with Flan, a Datalog compiler fully embedded in Scala that leverages multi-stage programming to generate specialized code for enhanced performance. Our approach combines the flexibility of Flix with Soufflé's performance, offering seamless integration with the host language that enables the addition of powerful extensions while generating specialized code for the entire computation. Flan's simple operator interface allows the addition of an extensive set of features, including arbitrary aggregates, user-defined functions, and lattices, with multiple execution strategies such as binary and multi-way joins, supported by different indexing structures like specialized trees and hash tables, with minimal effort. We evaluate our system on a variety of benchmarks and compare it to established Datalog engines. Our results demonstrate competitive performance and speedups in the range of 1.4× to 12.5× compared to state-of-the-art systems for workloads of practical importance.

CCS Concepts: • **Theory of computation** → **Program analysis**; *Constraint and logic programming*; *Logic and databases*; • **Software and its engineering** → *Automated static analysis*; *Compilers*.

Additional Key Words and Phrases: Datalog, Logic Programming, Generative Programming, Program Analysis

ACM Reference Format:
Supun Abeysinghe, Anxhelo Xhebraj, and Tiark Rompf. 2024. Flan: An Expressive and Efficient Datalog Compiler for Program Analysis. *Proc. ACM Program. Lang.* 8, POPL, Article 86 (January 2024), 33 pages. https://doi.org/10.1145/3632928

1 INTRODUCTION

Datalog has experienced a resurgence in popularity due to its high expressivity and ease of use in various applications. Its simple and intuitive declarative nature makes it readily applicable in a range of domains, including business analytics [Aref et al. 2015], graph analysis [Aberger

Authors' address: Supun Abeysinghe, tabeysin@purdue.edu; Anxhelo Xhebraj, axhebraj@purdue.edu; Tiark Rompf, tiark@purdue.edu, Department of Computer Science, Purdue University, 610 Purdue Mall, West Lafayette, IN, 47907, USA.

ACM 2475-1421/2024/1-ART86
https://doi.org/10.1145/3632928

et al. 2017; Seo et al. 2015; Shkapsky et al. 2016], declarative networking [Loo et al. 2006], binary dissambly [Flores-Montoya and Schulte 2020], and declarative program analysis [Allen et al. 2015; Bravenboer and Smaragdakis 2009; Reps 1994; Scholz et al. 2016; Ullman 1989; Whaley et al. 2005]. In particular, in the field of declarative program analysis, Datalog has proven to be a valuable tool for implementing intricate analyses that would otherwise necessitate thousands of lines of imperative code. Specifically, numerous program analysis problems can be formulated as a form of fixed-point computations, which is precisely what Datalog is designed to handle.

While declarative program analysis simplifies matters significantly, earlier approaches utilizing Datalog were not as scalable and performant as their imperatively written counterparts. Datalog compilers [Sahebolamri et al. 2022; Scholz et al. 2016; Tielen 2023; Zhang 2020] have been developed to close this performance gap by generating specialized code for a given Datalog program. Among such Datalog compilers, Soufflé [Scholz et al. 2016] stands out as one of the most comprehensive systems with many person-years of engineering investment, having numerous features such as fast parallel specialized data structures [Jordan et al. 2019a,b, 2022], automatic index selection [Subotic et al. 2018], join order optimization [Arch et al. 2022], among others.

Although Soufflé generates specialized code for a specific Datalog program leveraging the idea of Futamura projections [Futamura 1971] (resulting in impressive performance), their style of code generation, as discussed in Section 2, sacrifices some potential for specialization and, hence, performance. Moreover, Soufflé exists largely as a monolithic, closed system. It accepts Datalog programs as input and produces specialized C++ code, which is then compiled into an independent binary. However, as identified in previous research [Bembenek et al. 2020; Sahebolamri et al. 2022], many Datalog programs , especially in the context of program analysis, do not exist in isolation, but rather as components of larger systems written in more comprehensive programming languages. While Soufflé offers mechanisms to interface with the generated binary, it suffers from the *two language problem* [Bezanson et al. 2014] where declarative analyses are written in a custom Datalog dialect and further computations not supported by Soufflé must be written in other languages with data stored in a common format. Moreover, this lack of interoperability with a host language becomes a bottleneck when efforts are made to incorporate extensions like user-defined lattices [Madsen et al. 2016; Sahebolamri et al. 2022], SMT constraints [Bembenek et al. 2020], etc.

At the opposite end of the spectrum, languages like Flix [Madsen et al. 2022] support first-class Datalog constraints and seamless embedding of Datalog logic within a general-purpose host language. This approach provides greater flexibility, allowing users to create modular Datalog programs that can be composed in various ways depending on the specific task. Additionally, it becomes feasible to enrich Datalog with features such as user-defined Lattices [Madsen et al. 2016], since the end user can employ a full-fledged language to define these data structures and associated operations. However, existing systems following this approach lack support for specialized code generation (akin to Soufflé), resulting in significantly lower performance when compared.

In this work, we aim to bridge the gap in designing Datalog engines that are both *flexible* (akin to Flix) and *performant* (akin to Soufflé). One might contend that this could be achieved by augmenting the textual frontend of Soufflé-like systems with capabilities akin to those of Flix. However, such an undertaking would be tantamount to constructing an entire programming language from the ground up, complete with a comprehensive type system and more. Additionally, significant effort would be necessary to expand the compiler backend in order to accommodate these new features, owing to the intricacies inherent to the backend engine.

In this paper, we investigate if it is possible to architect a Datalog system with three key characteristics: (1) generating fully specialized code, a critical factor for optimal performance; (2) creating a flexible backend that can easily accommodate a variety of features and evaluation strategies;

(3) creating a flexible frontend capable of seamless interaction with a full-fledged programming language, potentially using its features to enrich the frontend with various extensions.

In pursuing these objectives, we turned our focus towards established generative programming tools such as Lightweight Modular Staging (LMS)/Scala [Rompf and Odersky 2010], BuildIt/C++ [Brahmakshatriya and Amarasinghe 2021], and AutoGraph/Python [Moldovan et al. 2019]. These tools have demonstrated success in constructing various high-performance systems with runtime code generation capabilities [Abeysinghe et al. 2023; Brahmakshatriya and Amarasinghe 2022; Rompf and Amin 2019; Tahboub et al. 2018; Wei et al. 2020, 2019]. We selected LMS, one of the most well-established tools in this space, and used it to develop Flan — a Datalog compiler fully embedded in Scala that is capable of producing fully specialized code for any given Datalog program.

A key realization is that these tools (1) provide programmable, fine-grained code generation that allows full specialization of programs (2) facilitates seamless interoperability with host languages (e.g., Scala) in the frontend. In Flan, we leverage this interoperability with Scala to enrich our frontend with various features such as polymorphic rules, higher-order relations [Arntzenius and Krishnaswami 2016], user-defined lattices, and so on, by creating an embedding for Datalog in Scala (Section 5). Flan's implementation resembles a high-level Datalog interpreter implemented using multiple high-level abstractions (Section 3), while LMS effectively dismantles these abstractions during code generation, thereby eliminating any related runtime costs. Flan's design features a streamlined operator interface which enables effortless support for a host of features including user-defined aggregates, user-defined functions (UDFs), stratified negation, and more (Section 4.2). Moreover, this allows support for multiple execution strategies, such as multi-way and binary joins (Section 4.3), and different index structures like BTrees and Hash indexes (Section 4.4), with minimal effort. While the rest of this paper presents the system implementation within the Scala/LMS context, we believe that the same can be accomplished within other similar frameworks and languages like Python/AutoGraph and C++/BuildIt (technically, any language with operator overloading).

Contributions

- We elucidate the rationale behind our choice of employing generative programming, specifically, LMS, as the basis for constructing Flan (Section 2).
- We review essential background and demonstrate the construction of a simple Datalog interpreter from scratch in Scala (Section 3).
- We illustrate how to effortlessly transform the Datalog interpreter into a compiler that generates fast, specialized code by utilizing LMS (Section 4.1). We demonstrate the integration of various extensions such as constraints, UDFs, negations, and aggregations through streamlined abstractions (Section 4.2). We highlight Flan's backend flexibility by adding support for multiple join evaluation strategies and index structures (Sections 4.3 and 4.4).
- We showcase the Datalog compiler can be seamlessly embedded into Scala, capitalizing on existing features of the language (e.g., type system, abstractions) for enabling composable, polymorphic, higher-order Datalog programs. Moreover, we demonstrate how this simplifies the process of enriching Datalog with features such as user-defined lattices (Section 5).
- We evaluate the performance of Flan against existing Datalog engines such as Soufflé, Crepe [Zhang 2020], Ascent [Sahebolamri et al. 2022], and Flix [Madsen et al. 2016] on a set of benchmarks. These benchmarks include a relatively simple points-to analysis (formulated as relations), a comprehensive program analysis benchmark from Doop [Bravenboer and Smaragdakis 2009] applied to a selection of DaCapo benchmark programs [Blackburn et al. 2006a], and an analysis involving lattice-based reasoning [Lhoták and Chung 2011]. Flan consistently demonstrates competitive performance in comparison to state-of-the-art systems, achieving speedups in the range of 1.4× to 12.5× (Section 6).

```
// codegen logic for Loop
void visit_(/* elided */) {
  PRINT_BEGIN_COMMENT(out);
  out << "iter = 0;\n";
  out << "for(;;) {\n";
  dispatch(loop.getBody(), out);
  out << "iter++;\n";
  out << "}\n";
  out << "iter = 0;\n";
  PRINT_END_COMMENT(out);
}
```

Fig. 1. Code generation logic for fixed point loop in Soufflé. Uses non-hygienic string code templates that are syntactically ill-formed.[1]

```
                    // next-stage variables
var iter = 0        // : Rep[Int]
var cond = ...      // : Rep[Boolean]

while (cond) {      // staged loop because
                    // next-stage cond
  /*
  logic for loop body
  */
  cond = /* update condition */
  iter += 1
}
```

Fig. 2. Flan's fixed point loop code. Scala essentially serves as a macro system, ensuring code generation is syntactically well-formed and hygienic.

In Section 7, we examine the relevant literature, followed by drawing conclusions and discussing potential future work in Section 8.

2 WHY GENERATIVE PROGRAMMING?

In the previous section, we outlined our goal of creating a Datalog engine that delivers performance on par with existing compilers through specialized code generation while preserving the ability to extend both the frontend and backend. This section delves into these concerns, articulating the potential of a generative programming-based approach to effectively address these needs.

Specialization In the context of Datalog engines, specialization refers to the process of generating specific code tailored for the semi-naive evaluation of rules of a given program. There are various methods to achieve this specialization. Soufflé, a highly performant and well-regarded Datalog compiler in the field of program analysis, employs template metaprogramming for this purpose. Consider the code generation logic for a simple fixpoint loop operation. Figs. 1 and 2 present the corresponding logic in Soufflé and Flan, respectively. Soufflé's code generation is driven by an imperative IR derived from the given Datalog program. Specifically, it involves concatenating a set of pre-written, operator-specific, stringified code templates like the one in Fig. 1.

Granularity of Specialization This approach indeed attains a level of specialization, yet it leaves potential for additional specialization—and thus, performance—unexploited. Specifically, the translation happens at the granularity of the query operators (e.g., fixed points, search, insert, etc.). Hence, a lot of abstractions (index structures, etc.) remain in the generated code, which is harder for the downstream C/LLVM compilers to reason about. There are several approaches to push this specialization further. One approach is to use full progressive lowering using an extensive compiler with multiple IRs, as seen in tools like DBLAB [Shaikhha et al. 2016] or potential multiple MLIR dialects [Lattner et al. 2021]. However, this becomes only a partial solution if we need easy extensibility, as adding features would require notable changes to this IR, lowering passes, and so on. Alternatively, fine-grained programmable code generation using tools like LMS (Scala), BuildIt (C++), or AutoGraph (Python) could be a preferable choice, and has already proven effectiveness for SQL [Rompf and Amin 2019; Tahboub et al. 2018].

Lightweight Modular Staging LMS provides a way of controlling this specialization through types. Specifically, LMS introduces the concept of `Rep` types, representing computations that occur in the next stage and should, therefore, appear in the generated code. Computations (including control flow, etc.) that take place on regular types (e.g., `Int`, `String`, etc.) are evaluated at the current stage. For illustration, consider a power function written in LMS (left) and the corresponding

[1] https://github.com/souffle-lang/souffle/blob/9aca1614a8865476abd681f17544dc9032dbb186/src/synthesiser/Synthesiser.cpp#L557-L566

specialized code (generated in C) for `power(b, 5)` (right). Notice that `b` is a `Rep`-typed variable, indicating that it is a next stage variable, and hence, any computations performed on `b` (in this case, multiplications) should appear in the generated code.

```
// Scala LMS code
def power(b: Rep[Int], n: Int): Rep[Int] =
  if (n == 0) 1
  else b * power(b, n-1)
```

```
// Specialized code for power(b, 5)
int power5(int b) {
  return b * b * b * b * b
}
```

The Scala program on the left is a program written using LMS's `Rep`-types and LMS will automatically partially evaluate the program with respect to the given value of `n` and produce the residual program on the right. LMS operates by evaluating the program as a standard program for regular-typed values, while constructing a graph-like IR for operations involving `Rep` values [Rompf et al. 2012]. This IR goes through multiple optimizations like dead-code elimination (DCE), code motion, etc. [Bračevac et al. 2023; Rompf et al. 2013] and generates code in the target language.

This offers us a convenient means of controlling the level of specialization. For instance, if we were to utilize an off-the-shelf hash table for implementing our hash-based indexes needed for rule evaluation, we could employ `Rep[HashMap[K,V]]`. This implies that the `HashMap` abstraction will be present in the generated code (e.g., HashMap from the standard library). In fact, as discussed in Section 4.4, we adopt this strategy to incorporate Soufflé's BTree index structure [Jordan et al. 2019b] into Flan. However, if we had, instead, used a current stage abstraction, e.g., a `HashIndex` that uses `Rep[Array[V]]` internally, then this abstraction would not appear in the generated code. Instead, fully specialized versions of each method invocation (insert, contain, etc.), that uses native arrays would be present in the final code. In our experiments (Section 6.3), we discovered that such a fully-specialized index implementation can be an order of magnitude faster than a library-based, generic, off-the-shelf implementation.

Principled Code Generation Another advantage of using an existing tool like LMS to perform runtime code generation is that the operator logic can be completely decoupled from code generation, and implemented in a regular manner. For instance, as is well known, relying on pre-written code templates (like in Fig. 1) tends to be brittle, as the developer must manipulate these string fragments to produce the final code. These fragments provides no guarantee of syntactic well-formedness [Taha and Sheard 1997] and hygiene [Kohlbecker et al. 1986], leaving room for inadvertent variable capturing and name conflicts (e.g., reusing the same name `iter` in another template during `dispatch` call). Handling even minor syntactic details, such as curly braces, requires specific consideration. While this may appear relatively tractable in a simple setting, maintaining and coordinating such templates becomes increasingly intricate as the number of templates grows and is fundamentaly at odds with extensibility. In contrast, a key distinction in Flan's case (Fig. 2) is the handling of variable scoping, typing, etc. by the host language (Scala, in our case).

Flexibility All this results in an implementation that mirrors a relatively simple interpreter, constructed in a high-level language employing high-level abstractions, as exhibited in Section 3. This architecture becomes the key in achieving backend flexibility, given that these high-level abstractions enable the creation of a streamlined operator interface that is flexible enough to facilitate the addition of various features like user-defined aggregates, functions, negations, constraints, and so on, (Section 4.2) alongside support for different indexing structures (tree or hash-based; Section 4.4) and evaluation strategies (binary or multi-way joins; Section 4.3).

While this approach gives us a flexible and efficient backend, the question of crafting a flexible frontend remains. One possibility is to solely depend on a textual frontend and incrementally add functionality/extensions as required. However, given the demand for full-programming language like extensions, this would eventually result in an effort similar to creating a new language from scratch. A key realization is that we can use the interoperability with the host language of tools like LMS to our advantage. Building on this concept, and inspired by lots of prior work on embedded logic

```
  y := &x      pointsTo(y, x) :- addressOf(y, x).                    (1)
  y := z       pointsTo(y, x) :- assign(y, z), pointsTo(z, x).       (2)
  y := *x      pointsTo(y, w) :- load(y, x), pointsTo(x, z), pointsTo(z, w).   (3)
 *y := x       pointsTo(z, w) :- store(y, x), pointsTo(y, z), pointsTo(x, w).  (4)
```

Fig. 3. Simplified points-to analysis rules written in Datalog (used as a running example in Section 3)

$c \in \mathsf{B},\ x \in \mathsf{Var}, R \in \mathsf{Rel}$

$t ::= c \mid x$

$a ::= R(t_1, \ldots, t_n)$

$r ::= a :- a_1, \ldots, a_n$

$P ::= r_1, \ldots, r_n$

```
/* Auxiliary definitions for Relation */
case class Schema(fields: Seq[Field])
class Relation(val name: String, val schema: Schema)
/* AST definitions */
type BaseTy = Boolean | Int | String
enum Term:
  case Const(v: BaseTy)
  case Var(s: String)

case class Atom(relName: String, args: Seq[Term]):
  def rel: Relation = ...

case class Rule(head: Atom, body: Seq[Atom])

type Program = Seq[Rule]
```

Fig. 4. Formal syntax for Datalog programs (left) and corresponding AST definition in Scala (right)

DSLs, in Flan, we devised an embedding for Datalog using standard Scala (Section 5). This utilization of Scala infrastructure significantly reduces the necessary engineering effort. A key distinction between our embedding and previous work lies in our integration with code generation — thus becoming an extensible, *compiled* embedded logic DSL. This strategy also reaps additional benefits, such as utilizing Scala's type system for automatic basic type checking, including appropriate use of variables in custom aggregates, user-defined functions and rules, etc.

3 A DATALOG INTERPRETER

An initial approach to developing a Datalog engine could involve converting a Datalog program into a collection of relational queries to be run on an SQL query engine, as explored by Scholz et al. [2015]. To implement the efficient semi-naive evaluation necessary for Datalog, we would need to: (1) represent each relation as three corresponding tables: '*base*', '*delta*', and '*next*' (2) establish an external driver loop to the query engine to enforce fixpoint semantics and manage records across the sub-relations. However, such trivial implementations usually perform more poorly than systems specifically designed for Datalog [Scholz et al. 2016] unless special care is taken [Fan et al. 2019; Ryzhyk and Budiu 2019]. The main challenge lies in creating and maintaining carefully selected indices (e.g., moving tuples from *next* to *delta*, etc.), and avoiding redundant computations arising from semi-naive evaluation. Nonetheless, it has been shown that building an optimized SQL engine can be done in 500 Lines of Code (LOCs) [Rompf and Amin 2015, 2019]. Although this engine cannot be repurposed directly due to the limitations mentioned, it prompts the question: Can we adopt similar principles to develop an efficient Datalog interpreter?

In this work, we demonstrate that the answer is indeed yes! Drawing inspiration from Rompf and Amin [2015]'s SQL engine, we will develop a succinct bottom-up Datalog interpreter in this section, utilizing semi-naive evaluation as illustrated in standard database textbooks [Ullman 1988, Figure 3.6] [Abiteboul et al. 1995, Algorithm 13.1.2]. For the sake of brevity, our initial focus will be the pure Datalog subset, excluding negations, aggregations, and other extensions. Nevertheless,

```
/* Entry point of the interpreter */
def compute(prog: Program,
            edbs: Map[Relation, Seq[Record]],
            outputs: Seq[Relation]): Map[Relation, Array[Record]] =
  val strata = stratify(prog)        // construct dependency graph of relations, find
                                     // strongly-connected components, and topologicaly sort
  val store: Store = Store()         // store: Stores tuples of relations
  for ((rel, recs) <- edbs)
    recs.foreach(store.insert(into=rel, _))          // Load EDB relation tuples into store

  for (rules <- strata)
    eval(rules)(store)                               // Compute IDB by evaluating rules

  outputs.map(rel => (rel, store.records(rel))).toMap // Collect outputs and return

/* Evaluating rules of a stratum */
def eval(rules: Program)(store: Store): Unit =
  val relations = rules.map(_.head.rel).toSet            // Relations of the stratum
  val (simpleRules, recursiveRules) = expandRules(rules) // Split into simple and recursive
                                                         // Add delta variants for recursive
  for (rule <- simpleRules)
    for (record <- evalRule(rule))                       // Evaluate simple rules
      store.insert(into=rule.head.rel, record)           // Project join output and insert

  while store.hasNextIteration(relations) do  // Evaluate recursive rules until fixed point
    for (rule <- recursiveRules)
      for (record <- evalRule(rule)(store))
        store.insert(into=rule.head.rel, record)         // Project join output and insert
```

Fig. 5. Main interpreter loop that drives rule evaluation. Function `compute` partitions the relations and rules into strata, and calls `eval` to evaluate each stratum.

the integration of these features is fairly straightforward and will be discussed in Section 4.2. We will use Scala 3 syntax throughout the paper.

3.1 Datalog

A Datalog program consists of a set of clauses of the form $a :- a_1, a_2, \ldots, a_n$. Each clause is composed of a head atom a and body atoms a_i, where each atom possesses an associated arity n and a corresponding number of term arguments t_i, denoted as $R(t_1, \ldots, t_m)$. A term argument can be either a variable x or a constant c of a base type B. The head atom of a rule defines facts of a relation while body atoms form a conjunctive query over multiple relations constrained by a sequence of arguments. The predicate symbol R associated with the atom is referred to as the relation. There are primarily two types of relations: *Extensional* relations (EDB), the base relations that serve as the input and *intensional* relations (IDB), intermediate or output relations derived by applying the rules in the Datalog program.

We show the formal syntax and Scala abstract syntax definition in Fig. 4. The auxiliary definition `Relation` holds metadata of relations. A relation's facts are derived by the rules in which the relation appears in the head atom. A relation has a schema that defines its arity and the column names (`Fields`). The `rel` property performs name resolution and provides the relation an atom refers to.

Consider a basic program analysis task formulated using Datalog as shown in Fig. 3 [Smaragdakis and Balatsouras 2015]. Specifically, this is a simplified version for Andersen-style pointer analysis [Andersen 1994], which aims to determine the objects a variable may potentially point to [Smaragdakis and Balatsouras 2015]. This analysis involves four EDB relations: *AddressOf*, *PointsTo*, *Load* and *Store*. We will use this as a running example in the rest of Section 3. The rules are reasonably clear and self-explanatory. For instance, rule (1) indicates that if `y` holds the address of `x` (i.e., `y := &x`), then `y` points to `x`.

The actual evaluation of the clauses relies on fixed-point semantics. In this process, the rules are applied to facts, initially starting from EDBs and subsequently with IDBs as they are derived, until

$$
\begin{array}{lll}
pointsTo.\text{next}^1(y,x) & :- \ addressOf(y,x) & \text{(Rule (1) expansion)} \\
\\
pointsTo.\Delta^i & := pointsTo.\text{next}^{i-1} - pointsTo^{i-1} & \\
pointsTo^i & := pointsTo^{i-1} \cup pointsTo.\Delta^i & \text{(Rule (2) expansion)} \\
pointsTo.\text{next}^i(y,x) & :- \ assign(y,z), pointsTo.\Delta^i(z,x) &
\end{array}
$$

Fig. 6. Rule expansion for rules (1) and (2) from Fig. 3 performed by expandRules (called in Fig. 5 L19). The first rule's expansion is simple and can be evaluated only once outside the fixpoint loop while the second one is recursive and is decomposed into a rule over delta relations.

$$pointsTo(y,w) :- load(\overset{sym}{y}, \overset{ptr}{x}), pointsTo(\overset{src}{x}, \overset{dst}{z}), pointsTo(\overset{src}{z}, \overset{dst}{w}).$$

```
for (x <- uniqueValues(of=ptr, from=load,     having={}))
  for (z <- uniqueValues(of=dst, from=pointsTo, having={src: x}))
    for (w <- uniqueValues(of=dst, from=pointsTo, having={src: z}))
      for (y <- uniqueValues(of=sym, from=load,     having={ptr: x}))
        yield (y, w)
```

Fig. 7. Top: Datalog rule (3) from Fig. 3 with column names for each relation shown on top of each variable. Bottom: Corresponding multi-way join nested loop.

the evaluation reaches a fixed point. This method is referred to as naive evaluation. However, it is inefficient due to the presence of numerous redundant computations. For instance, in rule (2), any tuples of *PointsTo* found prior to the previous iteration would not result in new tuples during the current iteration (since *Assign* is fixed). Consequently, it is more efficient to operate only on the tuples discovered in the immediate previous iteration (referred to as the *delta*). This strategy of employing deltas for fixed-point computation is known as semi-naive evaluation and is regarded as the state-of-the-art evaluation algorithm for Datalog.

3.2 Stratification

We now proceed with the definition of the interpreter. The evaluation of a Datalog program P can be decomposed into the evaluation of subprograms P_i where $\{P_1, \ldots, P_n\}$ is a partitioning (stratification) of the rules in P [Abiteboul et al. 1995]. The partitioning is obtained by computing the topological sort of the Strongly connected components (SCCs) of the (cyclic) dependency graph defined by the head relation and body relations of each rule. Each SCC i is a set of relations that are mutually recursive and P_i is defined as the set of rules defining any relation in SCC i. For instance, for our running example, there would be five strata: one for each EDB relation (*addressOf*, *assign*, *load*, and *store*) and another for the IDB relation *pointsTo*. The last stratum contains all four rules. In Fig. 5 we show `compute`, which is the entry point of our Datalog interpreter. First the program given as input is stratified through `stratify` (L5), then records for each EDB are loaded into a `store` (L8 and 9) and finally the interpreter evaluates the rules for each stratum in the topological order.

3.3 Stratum Evaluation

Typically, a stratum comprises multiple rules, and the interpreter's core functionality lies in how these rules are evaluated, which corresponds to `eval` in Fig. 5. Stratum rules are partitioned into simple and recursive rules. Simple rules are non-recursive rules, i.e., the body does not mention any of the relations present in the current stratum, and therefore can be computed immediately outside of the fixpoint computation. Recursive rules, by contrast, need to be evaluated until a fixpoint is reached. The evaluation of a rule yields records that can then be stored as shown in L23 and 28.

```
1 /* body: remaining body, iters: remaining order, env: current environment */
2 def multiWayJoin(body: Set[Atom], iters: Seq[JoinOperand], env: Map[Var, Value])
3                  (store: Store) = new:
4   def foreach(f: Map[Var, Value] => Unit): Unit =
5     iters match
6       case Seq(op, rest*) =>
7         val (filter,missingFields) = lookup(op.atom,env)            // lookup bound fields
8         for (value <-store.uniqueValues(of=op.field,from=op.atom.rel,filter)) // iterate variable
9           val newEnv = env + (op.arg -> value)
10          val (readyIncl,remaining) =                        // remaining: has unbound columns
11            body.partition(lookup(_,newEnv)._2.isEmpty)      // ready: all columns bound
12          val ready = readyIncl - op.atom
13          if (ready.forall(a => store.contains(a.rel,lookup(a,newEnv)._1)))
14            multiWayJoin(remaining,rest,newEnv)(store).foreach(f) // perform rest of join
15    case _ => f(env) // join completed
16
17 /* main entry-point for rule evaluation */
18 def evalRule(rule: Rule)(store: Store) = new:
19   def foreach(f: Record => Unit): Unit =
20     val order = variableOrder(rule.body)          // compute the order to perform the join
21     for (env <- multiWayJoin(rule.body.toSet, order, Map())(store))
22       val (record, _) = lookup(rule.head, env)    // project to rule head
23       f(record)
```

Fig. 8. Code for evaluation of rules. First, the variable order is computed (`order`), followed by the invocation of `multiWayJoin` that iterates through variables in the specified order and produces the join output. Code in L8 highlighted in blue corresponds to the nested loops in Fig. 7.

To implement semi-naive evaluation, recursive rules need to be expanded by `expandRules` in L19 into rules operating on three sub-relations: *base*, *delta*, and *next*. The *base* sub-relation contains all records discovered prior to the current iteration, while the *delta* sub-relation includes only the records found in the previous iteration. The *next* sub-relation holds the records derived during the rule evaluations of the current iteration. Fig. 6 illustrates expansions for rules (1) and (2) in the points-to program from Fig. 3. For rule (1), the `expandRules` function results in one simple rule with only *addressOf* in the body, which can be evaluated once outside the fixpoint loop. For rule (2), it results in a rule with a join between *assign* and the delta relation of *pointsTo*. The notation $:=$ represents table updates performed by `store.hasNextIteration(relations)` (L25), which computes the stable, next, and delta relations for the new iteration, as shown by the set operations.

3.4 Rule Evaluation

Next, we will explore the evaluation of rules. Broadly speaking, Datalog rule evaluation strategies fall into two categories: binary joins and multi-way joins. While binary joins typically involve iterating through entire tuples from relations, multi-way joins function at the granularity of individual variables. Our full system, Flan is capable of accommodating both strategies with only slight modifications to the core evaluation logic (discussed in Section 4.3). For the sake of conciseness, this section will exclusively present and discuss the code for a multi-way join strategy.

As an example, let us consider rule (3) from Fig. 3, which performs a join across three relations predicated on the keys x and z (as depicted in Fig. 7). The bottom part of the figure illustrates an analogous 'for comprehension' that emulates the logic of join evaluation: `uniqueValues` yields the unique values of the field passed as argument to the parameter '`of`' from relation '`from`' where the remaining field are constrained by the '`having`' record (essentially a filter). For readers who are familiar with the Generic Join [Ngo et al. 2013] algorithm, this may look similar but without the intersections. We discuss how a similar notion to intersections is achieved in this setting via explicit contains checks in Section 4.3, but omit it here for brevity.

For a given rule, the initial step involves computing the order in which variables (along with their corresponding relations) should be iterated. Upon establishing this order, the `uniqueValues` method can be invoked from the pertinent relations in the determined order, as illustrated in Fig. 7.

```
case class Value(v: BaseTy)
case class Record(schema: Schema, values: Seq[Value])

class Store:
  val records =                                                  // records for each relation
   mutable.Map().withDefault((_:Relation) => Set[Record]()) // are maintained in Sets

  def insert(into: Relation, record: Record): Unit =         // inserting new records
    records(into) = records(into) + record
  def contains(rel: Relation, record: Record): Boolean =     // checks existance of records
    records(rel) contains record

  def hasNextIteration(relations: Set[Relation]): Boolean =
    var updated = false
    for (relation <- relations)
      // Perform the updates shown in (Rule (2) expansion) from Fig. 6
      records(relation.delta) = records(relation.next) diff records(relation.base)
      records(relation.base) = records(relation.base) union records(relation.delta)
      records(relation.next) = Set()
      updated |= records(relation.delta).nonEmpty
    updated // returns false when a fixed point has been reached

  def uniqueValues(of: Field, from: Relation, having: Record): Iterable[Value] =
    records(from).filter(rec => rec(having.schema) == having.values).map(_(of))
```

Fig. 9. Code for `Store`, which maintains tuples of relations and is used throughout the evaluation.

It is important to note that while the sequence of iterating over the variables does not influence the correctness of the computation, it can have a substantial impact on the execution time of the query.

For completeness, the implementation for rule evaluation is shown in Fig. 8. The entry point is `evalRule`, which computes a `variableOrder` (L20) and then calls `multiWayJoin` which corresponds to the evaluation of the nested for loops generating the unique values for each variable and introducing them in `env`. Specifically, L8-14 of Fig. 8 corresponds to evaluating the rule using a loop nest, such as Fig. 7. The loop nest iterates over each join variable, with relation lookups interspersed at appropriate places. `multiWayJoin` returns an object implementing `foreach` allowing to iterate over the joined records through a simple for comprehension. Finally the loop body `f` is called with the projection of the environment corresponding to the rule's head record.

Close attention should be paid to the conditional present in L13: whenever the introduction of a variable of the rule completes one of the rule's body atoms, an contains check takes place, essential to produce the correct output. For example if we had picked *ptr* from *load* to yield the possible values of x and *src* from *pointsTo* for the values of z, then we must also check that (x, z) is present in *pointsTo* as shown below.

```
for (x <- uniqueValues(of=ptr, from=load,     having={}))
  for (z <- uniqueValues(of=src, from=pointsTo, having={}))
    if (pointsTo contains (x, z))
      ...
```

In Flan, we mimic the join order (computed by `variableOrder`) of left-associative binary joins with join variables iterated first. However it is straightforward to extend the system to support other query plans based on user provided hints or heuristics like cardinality/selectivity estimation.

3.5 Record Store

The last key remaining piece is the `Store` which will store the records that are produced during evaluation, ensure uniqueness of records, and enable fast retrieval of the unique values present in a specific field of the relation. Its implementation is shown in Fig. 9.

The `records` map contains the records of a given relation. The `insert` and `contains` method allow to perform insertions of a record into a relation (Fig. 5, L9, 23 and 28) and check the presence of a record in a relation (Fig. 8, L13), respectively. `hasNextIteration` checks whether a fixpoint has been reached (Fig. 5, L25) and performs the $:=$ updates shown in Fig. 6. Finally `uniqueValues` iterates

over the unique values of a column of a relation on the specific subset of records satisfying the filter (`having`), as discussed in Section 3.4 and shown in Fig. 8, L8. The implementation of the `Store` presentend is deliberately simple and inefficient. We defer the discussion of design decisions of the `Store` to Section 4.4. In particular, we introduce the notion of `IndexedStores`, which employs various index structures such as trees or hash tables to execute the required operations efficiently.

Combining all of these components results in a relatively simple Datalog interpreter. Although easy to construct, its performance is significantly inferior to that of a compiled engine generating specialized code. This is due to the substantial runtime interpretation overhead, such as the overheads associated with high-level abstractions and using generic data structures for indices. In Section 4, we will explore how to transform this basic interpreter into a compiler capable of producing efficient specialized code, requiring only minimal modifications to the original code.

4 FLAN: DATALOG COMPILER

4.1 Deriving a Datalog Compiler from the Interpreter

In Section 3, we demonstrated how to construct a relatively simple Datalog interpreter in Scala. In this section, we explore how to transform our slow interpreter into a compiler that generates fast, specialized code with minimal modifications to the original interpreter. We will employ the concept of partial evaluation and Futamura projections [Futamura 1971] to achieve this goal.

Partial Evaluation The concept of partial evaluation [Jones et al. 1993] involves decomposing the evaluation process into two or more stages, often based on the availability of inputs, with each stage evaluated to generate a residual program that contains the logic for evaluating the remaining stages. Consider our interpreter as an example: it accepts a Datalog program and the actual input data (i.e., EDB) as inputs and produces an output.

```
output = interpreter(datalog_program, input)
```

In the context of program analysis, the same `datalog_program` implementing the analysis is applied to multiple different programs to be analyzed. Consequently, it is sensible to divide the interpreter into two stages. Initially, we partially evaluate the Datalog interpreter with respect to the given Datalog program, obtaining a staged interpreter specialized for that specific analysis. This process is facilitated by a program specializer (or partial evaluator). Ideally, the specialized code should eliminate any overheads associated with interpreting the Datalog program while maintaining the ability to handle dynamic input EDBs.

```
specialized = specializer(datalog_interpreter, datalog_program)
output = specialized(input)
```

The first Futamura projection [Futamura 1971] states that executing the interpreter produces the same output as evaluating the specialized (i.e., partially evaluated) code with the same input. Furthermore, it states that this process of specialization is analogous to compilation. In essence, the `specialized` code mentioned above would be equivalent to code generated by a Datalog compiler.

One way to achieve this specialization is to create a custom program transformation tailored to the use case (i.e., Datalog) that generates the required `specialized` code for a given input Datalog program. However, this can result in a notable engineering effort as it is essentially equivalent to writing a compiler from scratch. Instead, we can rely on an existing generative programming framework like LMS to do this in a more principled manner as discussed in Section 2.

Our Approach using LMS As mentioned in Section 2, LMS leverages a type-based approach to facilitate this stage distinction and specialization. To achieve this, it introduces the notion of `Rep`-types. `Rep`-typed variables (e.g., `Rep[Int]`, `Rep[Array[Long]]`, etc.) designate them as next-stage values. Consequently, any operations conducted on these variables would appear in the generated code for next stage. In contrast, operations on variables with regular types (e.g., `Int`, `Array[Int]`, etc.) are executed in the current stage. LMS takes regular Scala programs with `Rep`-type annotations

```
def stratify(prog: Program): Seq[Program] = ...
def compute(prog: Program,
            edbs: Map[Relation, Seq[Record]],
            outputs: Seq[Relation]): Map[Relation, RepBuffer] = ...
def eval(rules: Program)(store: Store): Unit = ...
def evalRule(rule: Rule)(store: Store) = ...
def multiWayJoin(body: Set[Atom], iters: Seq[JoinOperand], env: Map[Var, Value])
                (store: Store) = ...

case class Record(schema: Schema, values: Seq[Value])

case class Value(v: Rep[_])

class Store:        // staged store implementation
  val records =
    mutable.Map().withDefault((_: Relation) => RepIndexedBuffer())

  def insert(into: Relation, record: Record): Unit
  def contains(rel: Relation, record: Record): Rep[Boolean]
  def hasNextIteration(relations: Set[Relation]): Rep[Boolean]
  def uniqueValues(of: Field, from: Relation, having: Record): RepBuffer
```

Fig. 10. Deriving a compiler from the interpreter via mixed-stage Store. Shown in gray are the previous definitions of the interpreter from Figs. 5 and 8 which are left unchanged. The only changes needed are: (1) `Values` types are now `Rep[_]` instead of Scala base types, denoting second-stage values (2) the store is updated to an implementation of a second-stage store (we elide the implementation).

as input, partially evaluates it using the Scala runtime, and subsequently generates specialized C code for the residual program.

In Fig. 10, we illustrate the modifications necessary to transform the interpreter, discussed in Section 3, into a compiler using LMS. The segments of code that remain unaltered are depicted in gray. First, we should identify static data available at staging, or in this case, code generation time. The primary available component is the Datalog program which contains the relation definitions along with their corresponding schema, input/output specifications, etc. Consequently, at staging time, we can compute the program's strata, topological order, recursive and non-recursive rules, join orders, and so on. Hence, in the LMS-based staged interpreter, computations pertaining to these aspects are carried out at staging time, leading to specialized code that eliminates any runtime overhead associated with these operations.

The sole piece of information not available at the staging time are the actual facts (i.e., records) of the EDB relations. Consequently, the values and data structures associated with these should be marked as next-stage values (i.e., `Rep` types). In particular, we update `Value` to contain `Rep[_]` values denoting next-stage values. This, along with other fixes related to type errors that arise from this change, are sufficient to convert our interpreter into a compiler that generates specialized code.

Finally we now need to use a staged version of `Set[Record]` in the records store (`Store` in Section 3.5, since now it contains next-stage values) and update its interface accordingly. For example, `contains` checks will now return `Rep[Boolean]` and be present in the residual program. The concrete implementation of `Set[Record]` is done through `RepIndexedBuffer`. We elide the implementation and defer a thorough discussion of implementation considerations to Section 4.4.

LMS readily offers staging support for most primitive operations involving `Rep`-typed values. For instance, LMS provides out-of-the-box support for assignments, comparisons, and other operations on primitive `Rep[T]` values. Additionally, language constructs such as **`if`**, **`while`**, and **`for`** involving next-stage values are transformed into their staged counterparts using macros [Rompf et al. 2012]. For example, when a `Rep[Int]` is compared with another next-stage value or a regular integer, it results in a `Rep[Boolean]`. If a conditional depends on this boolean value, the corresponding if block will be lifted into a staged if, which will ultimately be included in the generated code.

```
/* evaluate UDFs, aggregates and return result */
def evalArg(/* ... elided ... */): Value =
 case v: Variable => env(v)
 case Constant(v) => Value(v)
 case UDFCall(str, args) => evalUdf(str, args)
 case agg: Aggregator => evalAggr(agg)

/* evaluate all ready constraints and return whether all satisfied */
def evalReady(ready: Seq[Litereal], env: Map[Variable, Value]): Rep[Boolean] =
 ready.filter{case _: Assignment => false case _ => true}.forall{
  case Negation(atom) => !store.contains(atom.rel, lookup(atom, env)._1)
  case a: Atom => store.contains(a.rel, lookup(a, env)._1)
  case bc: BinaryConstraint => evalBinaryConstraint(bc, env) }

/* process any ready components before moving to next join step */
def processReady(remaining: Seq[Literal], env: Map[Variable, Value])
                (f: (Seq[Literal], Map[Variable, Value]) => Unit)(store: Store) =
 val (ready, nextRemaining) =                          // ready: components with
      remaining.partition(allVarsAvailable(_, env)) //        all vars in env

 val readyAssignments =                                // assignments that will
      ready.collect { case a: Assignment => a }        // introduce vars to env

 if (evalReady(ready, env))                            // process all ready constraints
  readyAssignments.headOption match
   case Some(assign) =>                                // process UDFs, aggregates, and
    val value = evalArg(assign.right, env)             // put result into env, and repeat
    processReady(nextRemaining ++ readyAssignments.tail, env + (assign.left -> value))(f)
   case None =>
    f(nextRemaining, env)

def join(/* elided */) = new:
 def foreach(f: Map[Var, Value] => Unit): Unit =
  processReady(body, env) { (remaining, newEnv) =>  // process any available UDFs, aggreagtes
   /* remain unchanged from Fig. 8 */               // constraints, etc. prior to join step
   iters match
    case Seq(op, rest*) => ...}
```

Fig. 11. The required code updates in Flan to accommodate negations, constraints, UDFs, aggregates, etc. Components are processed in `processReady`, which utilizes `evalReady` for evaluating negations and constraints, and `evalArg` for managing constants, UDFs, and aggregates.

4.2 Beyond Pure Datalog

Up to this point, we have developed an interpreter that supports pure Datalog and demonstrated how to transform it into a compiler that generates efficient specialized code. However, pure Datalog is not adequate for most practical program analysis tasks. For instance, the Doop analysis, one of the benchmarks we will examine in Section 6, employs various extensions such as stratified negation, aggregation, UDFs, constraints, and more. In this section, we will discuss how to incorporate these features into our Datalog compiler. Integrating these features is relatively straightforward because we can utilize high-level Scala capabilities to implement them (as we saw in Section 3) without concern for code generation or any runtime overhead associated with the abstractions.

Stratified Negation The concept of negation in Datalog serves to enforce that a given relation does not contain a specific value. Consider the following example of a rule with negation:

```
R(x, y) :- S(x, y), !T(x, y)
```

This rule states that we should insert all records from S that are *not* present in T to R. To ensure safety and unique fixed-point semantics, negation must be stratified, which means that negation can only be applied to a relation that appears in a previous stratum (in the topological order) [Chandra and Harel 1985]. During rule evaluation, negations are translated into simple contains checks. Specifically, when all variable bindings of a negated atom become available, we can perform a `contains` operation using the relevant index.

Aggregations, UDFs, and Constraints The example shown below presents a simple rule that incorporates a combination of features including aggregates, constraints, and User-Defined

Functions (UDFs). The rule aims to find the count of objects that a variable may point to for variable names that match a given regex. Here, `match` is a UDF, `count {PointsTo(var, _)}` is an aggregate, and `match(..) = true` is a constraint.

```
PointsToCount(var, p_count):-
Vars(var),
match("<some-regex>", var) = true,
p_count = count {PointsTo(var, _)},
```

```
// canonicalized version of rule on left
PointsToCount(var, p_count):-
  Vars(var),
  #udf1 = true,
  p_count = #aggr1,
  #const1 := "<some-regex>",
  #udf1 := match(#const1, var),
  #agg1 := count {PointsTo(var, _)}
```

The common trait in the evaluation of these extensions is that each component is processed as soon as the required variables become available in the environment. Thus, all we need to do is to modify the join logic to initially handle these 'ready' components before proceeding with the remainder of the join. To keep the logic for evaluating these extensions simple and general, we introduce the notion of rule canonicalization. Specifically, we rewrite the rules so that atoms in their body contain only variables, while other argument types are converted into variable assignments. Additionally, aggregates and UDF calls are hoisted into assignment operations using new variables.

Figure 11 shows the updated `join` function that incorporates the `processReady` method for handling the extensions. It takes as arguments the current environment and the remaining body literals, partitioning them into two categories: `ready`, encompassing literals with all necessary variables present in the environment, and `nextRemaining`, which includes the rest. For instance, in the initial join call for the rule above, `#const1` qualifies as a ready literal as it doesn't require any additional environment variables. The function `evalReady` then executes any pending constraint checks (like negations, binary constraints), and only if all constraints pass, the evaluation of the rest of the rule proceeds. Note that `evalReady` yields a `Rep[Boolean]`, indicating that constraint evaluation occurs in the next stage and the logic will feature in the generated code.

We treat `Assignments` distinctively; they introduce variables into the environment that could potentially render some remaining literals 'ready'. Therefore, we process one `Assignment` at a time, repeatedly employing the aforementioned process until all 'ready' components have been handled. We omit the code for evaluating binary constraints (`evalBinaryConstraint`) which simply looks at the constraint type and evaluate left and right hand side and compare based on the constraint. We also exclude the code for calling UDFs (`evalUDF`), which simply retrieves the corresponding UDF from the UDF map (where user can register their UDFs) and call it.

Flan fully supports user-defined aggregates. All users need to specify are an initial value for the aggregate and an update function, having type `(Value, Value) => Value`. The aggregate's body is evaluated using the same `join` function, starting with an initial environment where all free variables are bound by the corresponding outer variables. This also allows arbitrary nesting of aggregates. For brevity, we have chosen to exclude the code for `evalAggr(agg)` but it largely reuses already existing methods like `variableOrder` to determine the join order for aggregate sub query, `join` to perform the join, and `evalArg` for evaluating arguments (e.g., UDFs) used inside the sub query.

One significant advantage of maintaining high-level engine code in the interpreter style is the relative ease of adding new features, as seen above. In contrast, other existing approaches may necessitate more extensive modifications, such as augmenting their intermediate-level IRs, adding logic to their code generators, and more. We evaluate this ease of implementation in Section 6.2.

Incorporating functionality in this manner not only simplifies the process but also automates certain optimizations that other engines perform as separate passes. For instance, existing approaches rely on transformation passes on their imperative IR for optimizations such as hoisting aggregates, hoisting if blocks (or predicate pushdown), and collapsing filters [Scholz et al. 2016]. In contrast,

our interpreter inherently integrates these optimizations by design. For instance, aggregates, constraints, and UDFs are computed as soon as their variable bindings become available, and execution is short-circuited as early as possible.

4.3 Join Strategies

We initially outlined the implementation of multi-way joins in Section 3.4 when we introduced the code for our interpreter. In this section, we will delve further into the specifics, illustrating how we can facilitate both binary and multi-way joins seamlessly. We evaluate the performance of these strategies later in Section 6.3.

A core aspect of executing Datalog programs involves performing joins. Most existing Datalog engines rely on *relation-at-a-time* joins [Madsen et al. 2016; Scholz et al. 2016; Tielen 2023], which entail performing nested loop joins by iterating tuples from different relations at each level. This approach is analogous to binary joins in traditional relational database management systems (DBMS), with the key difference being that intermediate relations are not materialized. However, as well known in DBMS research [Atserias et al. 2008], using this type of binary joins for multiple relations can be asymptotically suboptimal in some cases. An alternative approach is to use *variable-at-a-time* joins, which leads to a class of join algorithms called worst-case optimal joins (WCOJ) [Aberger et al. 2017; Atserias et al. 2008; Ngo et al. 2018; Veldhuizen 2014]. In essence, the asymptotic complexity of these join algorithms are bounded by the worst-case output size of the *final* result, as opposed to the worst-case size of the *intermediate* results, as seen in binary joins.

The effectiveness of join strategies is influenced by several factors, including the cardinalities of relations, join selectivity, etc. Given this, prominent DBMS engines offer a range of join evaluation methods, using certain heuristics to select the most appropriate one [Freitag et al. 2020; Wang et al. 2023]. This underlines the necessity for Datalog engines to incorporate an array of built-in join strategies. Bearing this in mind, we implemented both join types in Flan. Given the abstractions previously defined, this addition was relatively simple.

We can make Flan seamlessly support both strategies by modifying a few interface APIs. First, we need to update the `uniqueValues(of: Field, from: Relation, filter: Record)` function to generate unique values across multiple columns simultaneously, rather than one column at a time. We achieve this by changing the '`of`' parameter's type from `Field` to `Seq[Field]`. Then, we update the `variableOrder` logic, which determines the sequence in which variables should be evaluated for the join. For variable-at-a-time joins, this yields a sequence of `JoinOperands`, each containing a variable and the related field of atom used to enumerate the chosen variable's values. To accommodate binary joins, we merely need a variant of `variableOrder` that returns a sequence of `JoinOperands`, each including an atom and *all* its associated fields necessary for iteration. This alteration implies that, instead of iterating one field at a time, we would iterate all required fields from an atom in a single join step. This enables the support of both join strategies, without any changes to the rest of the system code. Specifically, we have defined a `trait Strategy` that comprises an abstract `variableOrder` method. Then, we provide two traits—`MultiWayJoinStrategy` and `BinaryJoinStrategy`—that can be seamlessly mixed-in as needed to activate the required strategy.

Intersections This multi-way join evaluation strategy we saw in Section 3.4 closely resembles the Generic Join algorithm [Ngo et al. 2018], with the notable exception of the use of 'intersections', which is crucial for achieving strong asymptotic guarantees and runtime performance. In Generic Join, the set of iterated values for a variable is determined by intersecting the sets of values from all relations containing that variable. We achieve a similar notion to intersections by performing checks to determine whether an introduced variable is unifiable in the rest of the rule body atoms, and short circuits the execution when it is not unifiable. In essence, when a variable is introduced, we examine the other relations to determine if that variable is present before proceeding with

the remaining loop nest. We have incorporated these checks in both binary and multi-way join strategies, and our experiments affirm that their inclusion leads to noticeable runtime performance improvements.

Fused Traversals Variable-at-a-time iteration, as employed in multi-way joins, may introduce suboptimal looping structures under certain conditions. Specifically, consider a loop nest that consists of consecutive variable-iterating loops, where each of these loops iterate over variables from the *same* relation without any constraint checks or intersections between them. An example would be a simple rule like $R(a, b) :- S(a, b, c)$, which leads to a loop nest with two loops iterating over `a` and `b`, respectively. In such cases, the more efficient approach is to utilize a 'fused' traversal that combines these two loops, enabling iteration over both `a` and `b` with a single lookup. Our multi-way join strategy incorporates this traversal fusion as a plan-level optimization.

4.4 Indices: BTree and Hash Indexes

In Section 3.5, we constructed a generic implementation of `Store` without any indexing. However, this approach is not practical as lookups (i.e., `uniqueValues`) and checks (i.e., `contains`) are performance critical operations during execution. Consequently, it is essential to construct efficient indices to support these operations. There are several data structures at our disposal to build these indices. As discussed in prior work [Aref et al. 2015; Freitag et al. 2020; Ngo et al. 2018; Veldhuizen 2014], an indexing structure resembling a logical trie (each level representing a field) is necessary for variable-at-a-time joins. This can be realized through the use of an actual trie [Veldhuizen 2014], or by employing alternative data structures such as hash tables [Freitag et al. 2020] or trees that efficiently manage each level of the trie.

BTrees Our `Store` abstraction is agnostic to the backend index data structure, allowing us to select any suitable backend data structure as long as it supports the required operations. Datalog compilers like Soufflé are already equipped with fast index structure implementations, specifically tailored to Datalog, which have demonstrated impressive performance across a broad spectrum of workloads [Jordan et al. 2019b]. Hence, we have written a custom wrapper for Soufflé's BTree and integrated it into Flan to execute the lookups and checks efficiently. In this instance, as the data structure abstraction appears in the generated code, we use type `Rep[SouffleBTree]` for indices. It is important to note that this means the granularity of specialization would not be pushed beyond the BTree abstraction in the generated code.

Although our initial tree-based index implementation yielded good performance, we wanted to explore potential benefits of hash indexes due to their superior asymptotic performance (constant versus logarithmic), despite the fact that Jordan et al. [2019a] have shown that Soufflé BTrees consistently outperform standard library-based hash table indexes across diverse micro-benchmarks and full program analysis benchmarks. We also experimented with using library-based hash table implementations (using `Rep[HashIndex[K,V]]`), and observed similar performance behaviors to what they have demonstrated. A key realization is that these library-based data structures are overly generic (i.e., have room for more specialization), which consequently lead to significant runtime overheads as evident from our experiments in Section 6.3.

Fully Specialized Indices An alternative approach to achieve full specialization is to build our own data structures that operate directly on low-level `Rep[T]` values (e.g., `Rep[Array[T]]`), pushing the specialization granularity beyond the data structures like `HashIndex`, `BTree`, etc. We can still utilize high-level Scala data structures for this implementation, but staging will automatically erase all abstractions and operations performed on current stage values (i.e., non-`Rep` typed), resulting in the desired full specialization in the generated code. For instance, hash indices implemented in this way are transformed into sets of native C arrays in the generated code, where all operations (e.g., lookups, inserts, etc.) are inlined whenever they are invoked. This manner of generating

fully-specialized code offers more opportunities for additional optimizations (e.g., CSE, loop fusion, etc.) by downstream general-purpose compilers (e.g., gcc -O3) compared to having generic function calls to perform the operations. Furthermore, this approach gives us greater control over the data structure implementation, enabling us to fine-tune it for our specific use case. For instance, instead of storing the actual full key of the filter, we can simply maintain an offset (i.e., tuple id) to the central record buffer that stores all the records for a given relation, resulting in memory savings (also useful in adding support for lattices as discussed in Section 5.2).

We implemented another performance-enhancement tweak, specifically aimed at optimizing the transfer of `next` tuples required upon conclusion of each fixed point iteration. The most common approach is to first insert the tuples from `next` into the `base` relations, swap `next` and `delta`, and clear `next` (for all indices) [Scholz et al. 2016]. While this works well for tree-based indices, whose size depends on the actual data stored, our initial experiments suggested that, in some cases, it can result in poor performance for constant-size indices like hash indices, whose size does not directly depend on the actual element count. Specifically, we would end up clearing a large region of memory even if the previous iteration produced very few tuples. For instance, after performing n fixed-point loops, there would be `memset` calls at the end of each iteration (i.e., n sets of `memset` calls), regardless of how many `next` tuples were produced in the prior iteration. In some cases, these calls can become a bottleneck. In our hash indices, we have opted for a different approach, clearing the underlying data structures only when they are nearly full, thus amortizing the cost of clearing the buffers.

Now the question is, if we do not clear the delta hash tables, how to retrieve the delta for the current iteration (needed for semi-naive evaluation) because now the delta hash tables contain delta tuples for multiple previous iterations. To address this, we store the fixed point iteration number at which a given value was inserted within the hash index structure itself. This allows us to store values for deltas of all iterations in the same index and recover the last delta needed for semi-naive evaluation. Consequently, `next` simply becomes a watermark in the relation's record buffer, and we avoid clearing large memory regions that would consume a significant amount of time, particularly when there is a large number of fixed point iterations where each iteration takes a smaller amount of time (i.e., produces very few `next` tuples).

As discussed in Section 6, this finely-tuned, fully-specialized hash-based index demonstrated impressive results, consistently outperforming the tree-based index structure mentioned above. Additionally, akin to our approach with join strategies, this is also implemented in a 'modular' manner. We can easily incorporate the required index type by mixing-in the corresponding trait (`BTreeStore`, `HashStore`), without necessitating modifications to any other part of the codebase.

4.5 Identifying Required Indices

So far, we enriched Flan with prevalent Datalog extensions, demonstrated the ease of implementing different join strategies, and explored multiple index structures. Yet, the identification of necessary indexes for operations, a crucial element intersecting all these, remains. Most existing systems execute this step by conducting an analysis pass on their imperative IR [Sahebolamri et al. 2022; Scholz et al. 2016; Subotic et al. 2018], but in our case, we do not construct such an IR.

One feasible approach is to simply create all possible indexes for all relations and rely on the dead-code elimination (DCE) pass of LMS to discard any superfluous indexes in the final generated code. This strategy would function as expected because the LMS IR meticulously tracks the effects, thus enabling precise DCE for high-level data structures [Bračevac et al. 2023]. However, this approach may not be optimal since it looks at each operation locally, and does not consider the possibility of index sharing, hence, not minimizing index creation globally. Alternatively, we could implement custom logic that computes the necessary indices based on the evaluation strategy.

However, in this case, we would have to write this analysis separately for each evaluation strategy and would need to modify it each time new features (e.g., fused traversals as discussed in Section 4.3) are introduced, leading to a possible increase in maintenance complexity.

In Flan, we opted for a different approach: we perform a 'dummy' evaluation pass of the program using a `DummyStore` in place of the standard `Store`. The `DummyStore` mirrors the `Store`'s API but, rather than performing actual evaluations, it tracks the types of operations performed on each relation. Further, we ensure it reifies, for instance, both branches of a conditional, by creating dummy nodes in the IR (which don't generate any code), and returning next-stage values (e.g., `Rep[Boolean]` for `contains`).

The process of gathering required indices via dummy evaluation happens in two stages. First, an instance of a `DummyStore` is created and the program is 'evaluated' using the provided execution strategy. Although this uses the actual evaluation logic (i.e., core Flan code), this is not an actual evaluation, instead, this only collects the required indices for each relation. For example, when encountering a call to `store.contains(R, record)`, it will note the need for an index to facilitate a contains check on relation R for the fields `record.schema`. Once this dummy evaluation completes, it will give us the set of required iter indexes (to support faster `uniqueValues`) and check indexes (to support faster `contains`) for each relation.

After recording the operations performed on each relation, we can then create the minimal set of indexes required for these operations taking into account the possibility of index sharing. Since this dummy evaluation pass simply 'runs' the program, it is adaptable to any evaluation strategy, join strategy, and so forth, making it unaffected by changes in other parts of the system.

4.6 Parallelization

We adopt an approach similar to Soufflé for parallelization [Jordan et al. 2019b]. Specifically, we partition the outermost loop and execute the remaining rule evaluation in parallel using `pthreads`. However, when a new tuple is found, it is crucial to ensure that inserts occur in a thread-safe manner. To achieve this, we augment our hash index implementation to support parallel inserts by utilizing a series of locks, with each lock corresponding to a region of the hash keys (similar to `StripedHashSet` described in [Vu 2011]). Alternatively, we could achieve thread safety by relying on atomic operations, specifically, atomic fetch-add and compare-and-swap. However, a comprehensive discussion of this implementation is beyond the scope of this paper and thus omitted. Furthermore, the experiments presented in Section 6 that involve parallel execution report the performance for the aforementioned lock-based implementation.

5 CIPOLLA: A DATALOG DIALECT AS EMBEDDED DSL IN SCALA

In Section 4, we have presented the core of Flan, elucidating its ability to yield high performance while offering a broad range of functionalities. Although the textual Datalog frontend we employed (akin to Soufflé) facilitates the use across a variety of cases, as indicated in Section 1, certain use cases mandate seamless interoperability with a fully-fledged programming language.

The declarative nature of Datalog and its fixpoint semantics make it ideal for writing program analyses in a concise and clearer manner than writing them in a general-purpose language. Moreover, when coupled with an efficient engine, these analysis can scale to extremely large programs and be easily parallelized [Bravenboer and Smaragdakis 2009; Scholz et al. 2016]. Nevertheless, a growing amount of works [Arntzenius and Krishnaswami 2016; Bembenek et al. 2020; Madsen et al. 2016; Ryzhyk and Budiu 2019] is showing that it is possible use Datalog in a more diverse set of analyses, when the language is extended with additional features. In particular, Madsen et al. [2016] have identified the lack of support for UDFs and lattices as significant obstacles in applying Datalog to a wider range of dataflow analysis problems. Additionally, the lack of integration with existing

```
class Atm: // body of rules
  def ||(o: Atm): Atm = ...
  def &&(o: Atm): Atm = ...
  def where(cs: Constraints*): Atm = ...

type R1[T1] = Function1[T1, Atm]
type R2[T1, T2] = Function2[T1, T2, Atm]
// R3, R4, ...

def input[T1: Typ, T2: Typ](fname: String): R2[T1, T2] = ... // input relations

def rel[T:Typ](f: T => Atm): R1[T] = ...
def rel[T1:Typ, T2:Typ](f: (T1, T2) => Atm): R2[T1, T2] = ...// relations of different arities
... // other overloads

def exists[T:Typ](f: T => Atm): Atm = ...                    // introduction of vars
def exists[T1:Typ, T2:Typ](f: (T1, T2) => Atm): Atm = ...    // (see Fig. 12b for an example)
... // other overloads
```

(a) Definitions for implementing Datalog-like declarative rules using standard Scala constructs.

```
.type Id

.decl edge(src: Id, dst: Id)
.input edge

.decl path(src: Id, dst: Id)
.output path

path(src, dst) :- edge(src, dst).
path(src, dst) :- edge(src, node),
                  path(node, dst).
```

```
type Id = Rep[Int]

def edge: R2[Id, Id] = input("Edge.facts")

def path: R2[Id, Id] =
  rel { (src: Id, dst: Id) =>
    edge(src, dst) ||
    exists { (node: Id) =>
      edge(src, node) && path(node, dst)
    }
  }
```

(b) Expressing path Datalog program (left) in our Scala embedding (right)

Fig. 12. The core of our embedded DSL in Scala for writing Datalog programs, with (a) illustrating the definitions, and (b) displaying a sample program.

ecosystems complicates and renders interacting with Datalog computations inefficient, necessitating serializing queries and input/output data back and forth. To overcome these, they propose Flix, a full-fledged programming language with *first-class* support for Datalog. Bembenek et al. [2020] demonstrate that combining Datalog with Satisfiability-Modulo Theory (SMT) solvers enables writing more advanced and optimized SMT-based analyses. They propose Formulog, an extension of Datalog featuring a first-order fragment of ML and support for SMT formulas.

On one side we have highly-optimized Datalog query engines, where extensions can only be implemented by compiler writers. On the other, we move closer to general-purpose programming languages featuring Datalog-like declarative subsets, which prioritize modularity and abstraction at the expense of performance and require major engineering efforts to design and develop.

In this section, we reconcile these requirements, by embedding Datalog rules into LMS enabling abstraction without performance costs through *linguistic reuse* [Rompf et al. 2012] of host language (Scala) features. We will use concrete examples that have motivated the design of the Flix language [Madsen et al. 2022] throughout the section.

5.1 Embedding in LMS Scala

As a baseline, one can use the interpreter/compiler presented in Section 4 as a separate process from the main program. In this case, whenever the main program needs to evaluate a Datalog query, it can invoke the Datalog compiler as an external solver, sending the program in textual format along with the required data. The engine then compiles and evaluates the query, producing the result and sending it back to the main program. But this has a lot of overhead due to the repeated back and forth communication. Writing the main program in LMS instead already gives us some benefits — evaluating a Datalog program can easily be done by calling the staged version of `compute` from

Fig. 5 as `compute(parse(program), edbs, outputs)`. Code producing `edbs` and postprocessing the `outputs` can be optimized jointly with the staged interpreter code.

Yet, adding support for lattices and other extensions still requires designing additional syntactic constructs and typing rules. Instead, in Flan, we delegate these responsibilities to the host language, by providing an embedding of Datalog rules into Scala, similarly to Scalogno [Amin et al. 2019]. Fig. 12a shows how we designed this Datalog embedding (named Cipolla) in Scala that we use as a front end for Flan. An example Datalog program computing node reachability, as expressed through this embedding, is shown in Fig. 12b. In this setup, rules are formulated as functions wherein the arguments serve as the rule's metavariables, and the function bodies represent clauses. The `rel` combinator introduces relations of varying arities (`R1`, `R2`, etc). For instance, in Fig. 12b L6, `rel { (src: Id, dst: Id)...` defines `path` as a relation comprising two fields, `src` and `dst`. The argument function must produce an atom (`Atm`). Atoms are formed by applying relations — binding the variables of the defining relation to other relations (e.g., L8 `edge(src, dst)`), or by the disjunction (`||`) and conjunction (`&&`) of other atoms. These can also include constraints as specified by `where` (examples involving such constraints are provided later). The `exists` function introduces existential variables (variables that only appear on the right-hand side of a Datalog rule) explicitly. Additionally, we employ a `Typ` typeclass to ensure that relations are only defined for types supported by LMS's `Rep` types. From this point forward, we omit this typeclass constraint. We provide a `query` function that reifies the query into the AST format we used earlier for our textual frontend (Fig. 4).

```
val reachable = query[Int, Int] { (src, dst) => reach.path(src, dst) }
```

This reification process is done by repurposing the LMS staging capabilities. Essentially, we stage the program expressed in our embedding and construct the corresponding LMS IR. However, instead of using LMS backend for code generation from this IR as we typically do, we extract the LMS IR and transform it into our Datalog AST representation. Once the AST is derived, the subsequent steps of the process follow the same as described in Section 4. Specifically, in the above code, `query` will carry out the fixpoint computation and produce a second-stage buffer that can be used to print or further process the pairs of reachable nodes.

This embedding already allows us to reap multiple benefits from a developer experience perspective. We inherently gain the perks of syntax highlighting, refactoring, and typing, facilitated by the host language's tooling. Scala's type system automatically checks for ill-typed variable bindings in atoms within the rules. Going a step further, we can reuse other features of the host language such as case classes, pattern matching, polymorphism, and abstract methods when defining our program, as we shall discuss below.

Polymorphism and Typeclasses The program below generalizes the `path` program to work on any data-type `T` as long as a typeclass for `Eq` is available. `Eq` signifies that values of type `T` should have an equality function enabling comparison with other `T` values, which is essential for unification during rule evaluation. For example, `PolyReachability[Int]` would give us an instance of reachability that operate on graphs with `Int` labeled edges. Another key thing to note below is that the `edge` relation is left abstract such that users can provide different facts produced by means other than loading from a file, e.g., as result of a different rule.

```
trait PolyReachability[T: Eq] extends DatalogModule:
  def edge(src: T, dst: T): Atm
  def path: R2[T, T] = rel { (src: T, dst: T) =>
    edge(src, dst) ||
    exists[T] { node =>
      path(src, node) && edge(node, dst)
    }
  }
```

We achieve the benefits of mixing declarative programming with conventional functional programming as proposed by the Flix language [Madsen et al. 2022] leveraging host language abstractions. Moreover, by using LMS's `Rep` types we achieve 'abstraction without regret' [Rompf and Odersky 2010]: polymorphism and typeclasses are compiled away during staging and only code essential to the computation is generated.

Higher-order Relations Moving forward, it becomes evident that the definition of `path` is the transitive closure of `edge`. Therefore we can go ahead and rewrite our rules in terms of other higher-order rules, similarly to how it can be done in Datafun [Arntzenius and Krishnaswami 2016] and Scalogno [Amin et al. 2019].

```
def compose[A, B, C](r: R2[A, B], t: R2[B, C]): R2[A, C] =      // Compose Rule: (simple join)
  rel { (a: A, c: C) =>                                        // R(a,c) :- r(a,b), t(b,c)
    exists[B] { b => r(a, b) && t(b, c) }
  }
def transitive_closure[T](r: R2[T, T]): R2[T, T] =             // defines transitive closure
  rel { (x, y) =>                                              // using compose (above)
    r(x, y) ||                                        // tc(x, y) :- r(x, y)
        compose(r, transitive_closure(r))(x, y)       // tc(x, y) :- r(x, b), tc(b, y)
  }
def edge = ...
def path = transitive_closure(edge)                          // path: transitive closure of edge
```

The `compose` method becomes a shorthand for joining two relations and `transitive_closure` enables a more concise definition of `path`. Higher-order relations like the ones above enables us to rewrite rule (2) from the points-to example as follows.

```
def pointsTo = rel { (x, y) =>
  addressOf(x, y)                || /* rule (1) from Fig. (3) */
  compose(assign, pointsTo)(x, y) || /* rule (2) from Fig. (3), which is a simple join */
  ...
}
```

User-defined Functions Finally, we show how UDFs are integrated into the DSL. Given that we reside within Scala, users can create their UDFs using familiar Scala syntax. The following snippet shows an example where a UDF is used. Specifically, it shows a variant of the previously discussed 'path' program, wherein the `edge` has an extra attribute - `label`. The UDF serves as a predicate constraint, determining whether a given edge, based on its label, should be included in the path.

```
trait PolyReachability[T, L] extends DatalogModule:
  def edge: R3[T, L, T]

  def pred: L => Rep[Boolean]                  // abstract declaration of the UDF

  // path(src, dst) :- edge(src, dst, lab), pred(lab) = true
  // path(src, dst) :- path(src, node), edge(node, lab, dst), pred(lab) = true
  def path = rel { (src: T, dst: T) =>
    exists[L] { lab => edge(src, lab, dst) where pred(lab)} ||
    exists[T, L] { (node, lab) => path(src, node) && edge(node, lab, dst) where pred(lab) }
  }

val reach = new PolyReachability[Int]:
  def edge = ...
  def pred = (lab: Rep[Float]) => lab > 0.5     // UDF implementation

val reachable: Buffer[Int] = query { (dst: Id) => reach.path(2, dst) } // query
```

Since the query performed looks for paths from a specific `src` (2 in this case), at staging time, the concrete value of `src` will be used to generate the specialized code. Additionally the predicate function is inlined in the fixpoint loop in the generated code, minimizing function call overhead. As noted before, polymorphism is monomorphized during staging.

5.2 User-defined Lattices

With the aforementioned tight integration with the host language, adding extensions such as user-defined lattices becomes trivial. Essentially, we can use regular Scala classes to define lattices,

[2] https://doc.flix.dev/lattice-semantics.html#using-lattice-semantics-to-compute-shortest-paths

```
type Node = Rep[Int]
/* '@udd' indicating a lattice value */
@udd
case class D(x: Rep[Int])
/* defines lub - used for updates */
val lat = new JoinLattice[D]:
  def lub = udf { (x: D, y: D) =>
    val D(n1) = x
    val D(n2) = y
    D(n1 min n2)
  }
/* UDF on lattice values */
def add = udf { (x: D, y: D) =>
  val D(n1) = x
  val D(n2) = y
  D(n1 + n2)
}

def edge =
  input[Node, Rep[Int], Node]("Edge.facts")

def dist(src: Node): R2[Node, D] =
  rel { (dst, d) =>
    dist(src, D(0)) ||
    exists { (d1: D, d2: Rep[Int], x: Node) =>
      dist(x, d1) && edge(x, d2, dst) where
        (d `=` add(d1, D(d2)))
    }
  }
```

```
struct edge* edge_store = ...
/* index related data structures */
struct edge_ind* ind = ...
...

// fixed point loop
bool changed = true;
while (changed) {
  /* outer loops elided */
    // x, d1, d2, to in context
    // inlined udf
    int value = d1 + d2;

    // lattice update logic
    int oldValue = ...; // retrieve

    // inlined lub
    // values are unboxed
    int newValue = min(value, oldValue);

    if (oldValue != newValue) {
        // update the lattice ...
    }
}
```

Fig. 13. Code on the left demonstrates the implementation of the shortest path using lattices and UDFs, identical to Flix.[2] Unlike Flix, however, we generate highly specialized code that erases all high-level abstractions such as case classes, pattern matching, extractors, etc. A portion of the generated code is displayed on the right. Note that variables have been renamed for improved clarity.

regular Scala functions to define operations like least-upper bound for lattices, and regular Scala functions to define any UDFs operating on lattices. We examine a relatively simple example of using lattices alongside Datalog in Fig. 13 (left), which computes the shortest path from a given source node to all other nodes in the graph (taken from the Flix documentation).

While it is possible to encode the same problem in regular Datalog (that uses the powerset lattice) followed by an aggregation, this approach would be significantly slower, as it computes all possible distance values for each pair rather than just the shortest distance. Instead, as demonstrated in Fig. 13 (left), we can encode this problem using a user-defined lattice (L3) that maintains only the shortest path observed so far (similar to Flix). Note the use of `@udd` in L3, a macro we employ to track lattice types. Specifically, the `value` field of `dist` will now be a lattice value, and instead of retaining all previously seen values, it will only store the smallest value (defined using the `lub` function in L 6) observed so far for a given '`to`' value. Flan's backend is modified to handle lattice relations (i.e., ones having a `@udd` typed column) separately, and to call the corresponding `lub` function whenever new tuples are discovered. Moreover, whenever a new tuple triggers an update, we maintain these in our delta indices to facilitate semi-naive evaluation.

Fig. 13 (right) displays an excerpt from the code generated by Flan for this Datalog program. A crucial observation is that none of the abstractions used in defining the program on the left, such as case classes, object extractors, UDFs, and so on, are present in the generated code. Instead, it consists solely of operations on low-level primitive data types, native arrays, etc. ensuring that no runtime costs are incurred for the abstractions employed when defining the Datalog program. In Section 6.5, we evaluate the performance of programs that use these user-defined lattices and observe that the code generated by Flan is significantly faster compared to other existing systems.

In this example, we saw a scenario in which an unbounded lattice `D` is employed to represent the shortest distance, with $\bot$ implicitly signified by non-existence. However, there are situations

where bounded lattices with an explicit representation of $\top$ are necessary. For instance, the strong-update program analysis benchmark [Lhoták and Chung 2011] used in Section 6.5 makes use of the constant propagation lattice, encompassing $\bot$, a constant, and $\top$. In such cases, one option for users is to define a member within the case class (which represents the lattice) to indicate whether the current value is $\top$ or $\bot$. This would translate to a C struct representation in the generated code. Alternatively, users can use special values (e.g., `INT_MAX` for $\top$) to represent these.

6 EXPERIMENTS

In this section, we first evaluate the engineering effort associated with Flan and then the performance of Flan in comparison to various state-of-the-art systems across a diverse range of benchmarks. Our primary objective is to demonstrate that the use of staging for transforming a high-level interpreter into a compiler successfully generates specialized code that achieves competitive performance with other sophisticated Datalog compilers.

Our evaluation consists of three primary benchmarks. First, we select an existing simple points-to analysis benchmark from Soufflé and use it to compare Flan against a variety of other established systems. Next, we delve into a more comprehensive program analysis benchmarks from the Doop framework [Bravenboer and Smaragdakis 2009]. Lastly, we explore a use case that combines lattice semantics with Datalog and evaluate Flan's performance against systems that support lattices.

6.1 Environment

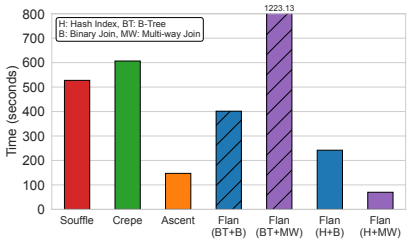
We conduct all our experiments on a NUMA machine featuring 4 sockets, each with 24 Intel(R) Xeon(R) Platinum 8168 cores and 750GB RAM per socket (3 TB in total), running Ubuntu 18.04.4 LTS. For multi-threaded executions, we employ `numactl` to ensure that processes are allocated within the same NUMA node and utilize the nearest memory regions. Each experiment is run five times, and we report the average across those five runs. We did not observe any significant variance among each run for all systems, hence, we omitted the error bars from the plots. For all experiments we have verified that all systems produce the same result. We use the most recent releases of all baseline systems, specifically: Soufflé 2.3, Ascent 0.5.0, Crepe 0.1.8, and Flix 0.35.

6.2 Engineering Effort

Since we consistently emphasize Flan's implementation simplicity throughout the paper, we now try to evaluate this aspect of Flan. While it is very hard to quantify the engineering effort scientifically, we have chosen lines of code (LOC) as a crude proxy for this evaluation. The table below illustrates the additional LOC required for each extension. 'Base' refers to the staged version of the interpreter initially introduced in Section 3, plus parsing, embedding (Section 5), other utilities like profiling (e.g., measuring per-rule time), logging, output verification, etc.

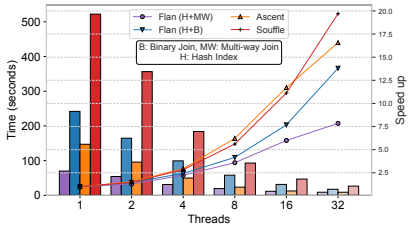
Base	+Features	+Join Strategies	+BTree	+HashIndex	**Total**
2197	+366	+130	+332	+1063	4088

The '+Features' case encompasses the integration of user-defined aggregates, UDFs, negations, constraints, etc. The added code pertains to the canonicalization process discussed in Section 4.2, and the logic highlighted in Fig. 11. This also includes the addition of support for user-defined aggregates, a feature recently added to Soufflé. Implementing this in Soufflé required over 1500 LOC modifications, impacting their declarative and imperative IRs, core engine mechanics, and code synthesizer [Henry 2022]. In constrast, we accomplished the same functionality with fewer than 50 lines of high-level Scala code. It is important to note, however, that this comparison is not rigorously precise, as LOC do not always accurately reflect implementation complexities.



Time (s)	Soufflé	Crepe	Ascent	Flan BT+B	Flan BT+MW	Flan H+B	Flan H+MW
PointsTo	527.50	606.64	147.16	401.45	1223.13	241.96	**69.97**
Speedup	1.00	0.87	3.58	1.31	0.43	2.18	7.54

Fig. 14. Performance comparison between Flan (our system, with different strategies) and various other Datalog compilers for a simple call-insensitive, field-sensitive points-to analysis, for single-threaded execution.



Time (s)	1	2	4	8	16	32
Flan (H+MW)	**69.97**	**54.27**	**31.29**	**19.48**	**11.67**	8.92
Flan (H+B)	241.96	164.68	99.37	58.12	31.49	17.53
Ascent	147.16	95.78	49.93	23.65	12.56	**8.88**
Soufflé	522.18	356.55	183.87	93.04	46.93	26.51

Fig. 15. Performance of parallel execution for the same benchmark in Fig. 14, comparing Soufflé and Flan (for binary and multi-way joins). The lines represent the scaling of each system relative to its own single-threaded performance.

The '+Join Strategies' denotes the addition of support for multi-way and binary joins, achieved by creating two implementations of the `trait Strategy` with the `variableOrder` overridden, as detailed in Section 4.3. '+BTree' and '+HashIndex' refers to the LOC added to implement the respective index structures. As outlined in Section 4.4, for BTree, we simply created a wrapper around Soufflé's B-trees, whereas for HashIndex, we developed our own specialized version in Scala.

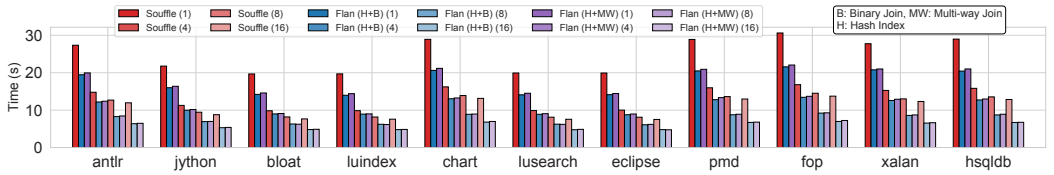
6.3 Simple Points-to Analysis

To evaluate performance relative to several other systems, we chose a relatively simple, openly available benchmark from Soufflé[3] that conducts a call-insensitive, field-sensitive points-to analysis using Datalog on a provided synthetic dataset. This analysis includes roughly 7,000 facts each in four EDB relations, two IDB relations, and five rules. We re-implemented these benchmarks in Ascent and Crepe, whereas for Flan and Soufflé, we directly input the corresponding Datalog file (Flan supports running Soufflé-style Datalog files directly). It is worth noting that this benchmark does not incorporate any extensions such as UDFs, aggregations, constraints, negations, or others.

Fig. 14 presents the benchmark results. Flan incorporates both binary and multi-way joins, while the rest of the baselines employ solely binary joins. Soufflé utilizes tree-based indices, which, compared to the hash-based indices used by all other systems, has a higher asymptotic runtime leading to longer execution times. Despite Crepe's use of hash-based indices, it lags noticeably behind Soufflé due to its reliance on generic hash table implementations, in contrast to Soufflé's data structures tuned for Datalog. Although Ascent generally uses binary joins, it employs a special optimization for certain rule types. This optimization transforms the outermost join steps into a form identical to multi-way joins. For instance, the performance-critical rules in this benchmark, which account for 53% and 44% of total time, contain only two atoms in the body (each with two variables), and performs a transitive closure. For these two rules, Ascent's optimization results in a join strategy identical to our multi-way joins, demonstrating a speedup of 3.25× over Soufflé.

The different execution strategies deployed in Flan showcase the distinct performance characteristics of each. For both strategies, our specialized hash-index offers substantial performance gains over their generic B-tree counterparts, yielding speedups of 17.5× and 1.66×. For this benchmark,

[3] https://github.com/souffle-lang/benchmarks/tree/acd6b9ec5043109fc1e6674e759a86164634e70b/benchmarks/pointsto



Time (s)	antlr	jython	bloat	luindex	chart	lusearch	eclipse	pmd	fop	xalan	hsqldb
Soufflé (1)	27.35	21.77	19.67	19.70	28.93	19.92	19.92	28.90	30.62	27.76	28.99
Flan (H+B) (1)	**19.46**	**15.99**	**14.21**	**13.98**	**20.62**	**14.10**	**14.14**	**20.48**	**21.57**	**20.78**	**20.44**
Flan (H+MW) (1)	19.95	16.38	14.59	14.40	21.17	14.51	14.45	20.92	22.06	21.00	21.01
Soufflé (4)	14.78	11.27	9.83	9.86	16.22	9.89	10.01	15.97	16.81	15.27	15.83
Flan (H+B) (4)	**12.19**	**9.98**	**8.99**	**8.94**	**13.04**	**8.89**	**8.77**	**12.80**	**13.43**	**12.59**	**12.72**
Flan (H+MW) (4)	12.38	10.16	9.10	9.00	13.25	9.05	8.97	13.34	13.70	12.91	12.98
Soufflé (8)	12.68	9.45	8.18	8.17	13.90	8.12	8.11	13.63	14.52	13.01	13.55
Flan (H+B) (8)	**8.27**	**6.93**	6.30	6.23	**8.89**	6.25	**6.09**	**8.76**	**9.21**	**8.59**	**8.73**
Flan (H+MW) (8)	8.45	6.98	**6.24**	**6.17**	8.96	**6.21**	6.20	8.91	9.28	8.72	8.91
Soufflé (16)	11.96	8.78	7.65	7.59	13.15	7.56	7.52	12.97	13.74	12.32	12.85
Flan (H+B) (16)	**6.37**	**5.31**	**4.81**	**4.79**	**6.80**	**4.77**	4.79	**6.72**	**6.98**	**6.54**	**6.71**
Flan (H+MW) (16)	6.48	5.39	4.88	4.85	6.98	4.87	**4.75**	6.80	7.24	6.64	6.75

Fig. 16. Performance comparison of our system (Flan) with Soufflé for the micro analysis variant of the Doop program analysis toolchain. The value within parentheses indicates (index structure + join strategy), followed by the number of threads used. The analysis is conducted on eleven programs from the DaCapo benchmark suite. The table presents the execution times for each scenario.

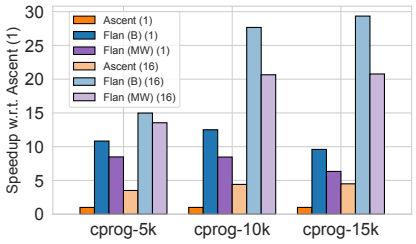
multi-way joins perform significantly better than binary joins. Upon closer inspection, we observed that the multi-way join strategy results in 23% fewer lookups compared to the binary join strategy (3,914,693 vs 5,126,129) for the performance-critical rules. Interestingly, the multi-way join case using B-trees notably underperforms compared to its binary join counterpart. This is because, despite a smaller number of lookups, the total cost of lookups is higher in the multi-way join case.

Although not shown in Fig. 14, we experimented with a variant of our multi-way join that used C++ standard library's unordered maps. The result was an execution time of over 30 minutes, an order of magnitude slower than our specialized hash-indexes. This disparity exemplifies the efficiency of having fully-specialized index implementations.

Parallel Scaling Fig. 15 shows the results for parallel execution performance. For this, we employed the same benchmark but varied the number of threads. Among prior systems, only Soufflé, Ascent, and Flan support parallel execution. Flan consistently outperforms Soufflé across all thread counts, achieving speedups ranging from 3× to 7.5× (compared to Soufflé with the same number of threads). Moreover, Soufflé requires 8 to 16 threads to match Flan's single-threaded performance. In comparison to Ascent, Flan demonstrates significantly better performance for up to 8 threads, achieving speedups of up to 2.1×, and remains competitive in 16 and 32 threads cases.

6.4 Doop: Points-to Analysis of Java Programs

We also investigated how each system's performance scales in relation to its single-threaded execution (depicted by a line graph in Fig. 15). All systems demonstrate good scalability as the number of threads increases, with Soufflé scaling to 22.6×, Ascent to 16.5×, and Flan reaching 13.8× (binary) and 7.8× (multi-way) at 32 threads. It is important to note that despite Soufflé's better scalability, it remains slower in absolute terms. Soufflé's better scalability can be attributed to their highly efficient concurrent index structures [Jordan et al. 2022] tailored for Datalog. Moreover, as identified by McSherry et al. [2015], systems with higher overall overhead tend to scale better due to the parallelization of overheads, which is likely another reason for Soufflé's superior scaling. Ascent uses the data parallelism-library Rayon [Rayon 2022] for parallel execution, which uses a work stealing model. For index structures, Ascent uses the DashMap library [Wejdenstål 2022], which is a highly-efficient concurrent hash table implementation in Rust. This allows Ascent to scale well as



Time (s)	cprog-5k	cprog-10k	cprog-15k
Ascent (1)	2.99	52.19	268.15
Flan (B) (1)	**0.28**	**4.17**	**27.95**
Flan (MW) (1)	0.35	6.17	42.36
Ascent (16)	0.85	11.84	59.66
Flan (B) (16)	**0.20**	**1.89**	**9.14**
Flan (MW) (16)	0.22	2.53	12.91

Fig. 17. Performance evaluation for performing a strong-update analysis (implemented using user-defined lattices) on three C programs. The left plot shows the normalized execution time with respect to Ascent single-threaded time. The table on the right presents the actual execution times for each case.

the number of threads are increased. In contrast, as discussed in Section 4.6, Flan's parallel data structures are currently based on relatively simple lock-based segmented hash tables. We believe the performance scalability can be further improved by utilizing more advanced concurrent index structures as we discussed in Section 4.6.

In the previous section, we evaluated performance for a relatively simple points-to analysis benchmark. In this section, we examine a more comprehensive benchmark, specifically choosing the micro analysis from the Doop program analysis benchmark suite. This analysis consists of 389 relations (including 109 EDB relations) and 300 rules, employing various Datalog extensions such as aggregation, negation, UDFs, and constraints. As done in prior similar studies [Antoniadis et al. 2017; Zhao et al. 2020], we run this analysis on a selected set of programs from the DaCapo benchmark suite [Blackburn et al. 2006a,b]. Doop extracts the facts for the EDB relations used in the analysis from these programs, which can then be used to perform the analysis using Datalog.

We evaluate the performance of Flan only against Soufflé, as none of the other systems directly support the analysis logic emitted by Doop or the various extensions used in the analysis logic. We evaluate both single-threaded performance and parallel performance, running on 4, 8, and 16 threads. We observe similar performance traits to those of the prior experiment. Both join variants of Flan utilizing specialized hash indices consistently outperform Soufflé. In particular, binary joins demonstrate an average speedup of 1.45× (across all cases) with a peak speedup of 1.97×. Meanwhile, multi-way joins show an average speedup of 1.42× and achieve a maximum speedup of 1.91×. It is worth noting that the binary join strategy in Flan differs slightly from Soufflé's since we also perform intersections to short-circuit the loop nest as early as possible (as discussed in Section 4.3). Although not included due to space limitations, we also conducted benchmarks on a variant of binary joins that does not perform these intersections, which resulted in an average speedup of 1.2× and a maximum speedup of 1.6× compared to Soufflé.

6.5 Strong-Update Points-to Analysis using Lattice Semantics

In this section, we evaluate the performance of a program analysis benchmark employing lattice semantics. More specifically, we selected the strong-update points-to analysis, as introduced by Lhoták and Chung [2011], which combines elements of flow-insensitive and flow-sensitive analysis. This is achieved by employing a singleton-set lattice to propagate singleton sets in a flow-sensitive manner. Our evaluation replicates the declarative implementation outlined in the work by Madsen et al. [2016] (Figure 4 in that paper), which was originally implemented using Flix. We have re-implemented this analysis both in Flan and Ascent.

We evaluate the performance of this benchmark on three sample C programs each having 5k, 10k, and 15k instructions, from which input relations `AddrOf`, `Copy`, etc. are extracted. Ascent has

previously demonstrated speedups of several orders of magnitude compared to Flix in similar benchmarks [Sahebolamri et al. 2022]. Consequently, we omit the Flix numbers for this experiment.

Fig. 17 illustrates the performance of Flan compared to Ascent for this benchmark. Specifically, it shows the normalized execution time with respect to Ascent's single-threaded execution time for both Flan and Ascent under single-threaded and 16-threaded execution settings. As in previous experiments, we report Flan's performance for both the multi-way and binary join strategies. In single-threaded execution, Flan demonstrates significant speedups, with the multi-way strategy achieving a maximum speedup of 8.5× and an average speedup of 7.8×. The binary join strategy in Flan attains a maximum speedup of 12.5× (average speedup of 11×) compared to the single-threaded performance of Ascent. When scaling up to 16 threads, Flan maintains its superior performance, with the multi-way join strategy delivering a maximum speedup of 4.7× (average speedup of 4.4×). Meanwhile, the binary joins in Flan achieves a maximum speedup of 6.5× (average speedup of 5.7×) when compared to their 16-threaded Ascent counterparts.

In this particular example, apart from the differences in join strategies as we discussed in previous subsections, the other main difference between Ascent and Flan is the specialization granularity. Ascent enables specialization for compiling the given Datalog rules, but any abstractions associated with defining lattices, UDFs, etc. will still be present in the generated code. Conversely, as seen in Fig. 13, Flan fully specializes the program, including any arbitrary programming logic surrounding the Datalog logic. In this case, this includes abstractions related to user-defined lattices and the corresponding UDFs. As a result, the generated code in Flan does not contain any of these abstractions or UDFs (inlined), thereby eliminating any runtime cost associated with them.

The superior performance of Flan's binary joins compared to multi-way joins in this specific benchmark can be attributed to the higher selectivities of variable intersections. In essence, this translates to a reduced amount of filtering or short-circuiting at each level within the loop nest in the multi-way case. As a consequence, a higher number of total lookups is conducted compared to the binary join case. It is important to note that this behavior can be highly dependent on the specific program and input data.

In summary, in this section, we have observed that Flan achieves competitive performance, and in some cases, significantly outperforms existing state-of-the-art systems across various types of workloads. This validates our claim that staging can be utilized to transform a high-level Datalog interpreter into a compiler that generates specialized code that achieves the same level of performance as other sophisticated Datalog compilers with only a fraction of the engineering cost.

7 RELATED WORK

Datalog for Program Analysis Datalog has been a popular choice for declarative program analysis for quite some time [Benton and Fischer 2007; Bravenboer and Smaragdakis 2009; Hajiyev et al. 2006; Hoder et al. 2011; Lam et al. 2005; Reps 1994; Smaragdakis and Bravenboer 2010; Ullman 1989; Whaley et al. 2005; Whaley and Lam 2004]. bddbddb employs Binary Decision Diagrams (BDDs) for evaluating Datalog rules; however, this representation is only effective for specific problem structures, and its performance is highly dependent on variable ordering. Soufflé [Scholz et al. 2016] is a high-performance Datalog engine that synthesizes specialized C++ code for a given Datalog program. As one of the most mature and widely used Datalog systems, Soufflé offers a broad range of features, including specialized parallel data structures [Jordan et al. 2019a,b, 2022], provenance [Zhao et al. 2020], automatic index selection [Subotic et al. 2018], incremental execution [Zhao et al. 2021], join order optimization [Arch et al. 2022], and more. Owing to its maturity, Soufflé has been used not only in popular program analysis toolchains like Doop [Bravenboer and Smaragdakis 2009; Smaragdakis and Balatsouras 2015] but also in other domains such as binary disassembly [Flores-Montoya and Schulte 2020] and decompilation of smart

contracts [Grech et al. 2019]. However, as discussed in Sections 1 and 2, there are several limitations in Soufflé's code generation approach. RecStep [Fan et al. 2019] is a Datalog engine built on top of the relational query engine QuickStep [Patel et al. 2018] and achieves competitive performance on diverse workloads including graph analytics and program analysis. However, similar to Soufflé, it lacks an expressive front end.

Several recent works have followed in Soufflé's footsteps in developing Datalog compilers. Crepe [Zhang 2020] and Ascent [Sahebolamri et al. 2022] are Datalog compilers embedded in Rust using macros and employing quasi-quotations for code generation of a given Datalog program. Eclair [Tielen 2023] is a Datalog compiler that translates Datalog programs into LLVM IR [Lattner and Adve 2004]. However, all these code generation approaches suffer from similar limitations to a certain degree as Soufflé (e.g., having to manage virtual registers when generating LLVM IR, writing quoted code fragments, as opposed to writing high-level interpreter-style code). Differential Datalog [Ryzhyk and Budiu 2019] (built atop Differential Dataflow [McSherry et al. 2013]) and IncA [Szabó et al. 2016] are Datalog engines that support incremental execution. Notably, IncA is tailored towards program analysis and has support for lattices [Szabó et al. 2018; Szabó et al. 2017]. Morever, IncA DSL has some superficial similarities to our Scala embedded DSL. Flan currently lacks support for incremental execution, an avenue we plan to explore in future work. Drawing upon previous work that maps e-matching [de Moura and Bjørner 2007; Willsey et al. 2021] into conjunctive queries [Zhang et al. 2022], EggLog [Zhang et al. 2023; Zucker 2022] unifies equality saturation with Datalog-style fixed point computations by baking in the notion of equivalence into Datalog relations. Exploring how to expand Flan to support equality saturation in a similar manner is an interesting future research direction. Both Zhang et al. [2022] and EggLog employ Generic Joins [Ngo et al. 2018, 2013], a type of multi-way join algorithm that dynamically picks the atom with lowest cardinality for variable iteration which leads to worst-case optimal joins.

BYODS [Sahebolamri et al. 2023], an extension to Ascent, offers a means to create user-defined data structures for storing relations which also capture properties (e.g., transitivity) of the relations. They have demonstrated significant performance improvements for certain types of program analyses using this approach. We believe that a similar notion can be realized in Flan in a similar way since Flan programs are fully interoperable with Scala, enabling users to employ Scala to define and integrate such custom data structures. However, it is important to note that we have not evaluated the plausibility or effectiveness of this approach in our current work.

Datalog Extensions Pure Datalog is not sufficiently expressive to cover a wide variety of declarative program analysis tasks, and many existing systems propose various extensions. Flix [Madsen et al. 2022, 2016] supports Datalog as first-class values in a relatively comprehensive programming language and supports lattices beyond the powerset lattice, allowing more expressive analysis like StrongUpdate [Lhoták and Chung 2011] analysis and, IFDS and IDE algorithms [Reps et al. 1995; Sagiv et al. 1996]. Datafun [Arntzenius and Krishnaswami 2020, 2016], another language that provides support for lattices also has the ability to track monotonicity via types. Formulog [Bembenek et al. 2020] extends Datalog with the ability to interact with a functional programming language and the capability to use SMT constraints in rules, offloading them to a solver as necessary. These approaches greatly enhance Datalog's power and applicability across various domains. However, most of these systems do not leverage techniques like specialized code generation, potentially sacrificing performance. We believe that Flan-style compiler construction, closely resembling interpreters, would make it easier for such extensions to achieve their optimal performance. Functional IncA [Pacak and Erdweg 2022] proposes the use of a functional programming with set-based fixpoint-semantics as a frontend for Datalog. In this model, Datalog serves as an IR, with programs written in the functional language translated into Datalog and resolved through pre-existing solvers. This approach contrasts with Flan's approach, where the 'general-purpose' aspect of the

computation is processed using the (general-purpose) host language, with the Datalog solver being specifically employed for fixed point computations.

Query Compilation Our work is primarily inspired by prior research in relational query compilation. Earlier such approaches [Freedman et al. 2014; Krikellas et al. 2010] relied on operator templates (similar to Soufflé) and generated code by concatenating these templates based on the query plan. Hyper [Neumann 2011], on the other hand, utilized the programmatic LLVM API [Lattner and Adve 2004], achieving significant performance gains but at the expense of a more low-level implementation. LB2 [Essertel et al. 2018; Tahboub et al. 2018] (an extension of [Rompf and Amin 2015, 2019]) was based on generative programming [Rompf and Odersky 2010] and achieved the same level of performance while keeping the operator interface simple. We draw inspiration from these works and adapt the same ideas to Datalog-based program analysis.

8 CONCLUSION

We introduced Flan, a Datalog compiler constructed by partially evaluating a high-level Datalog interpreter implemented in Scala. Utilizing the existing generative programming framework, LMS, we generate *fully* specialized code, a feature that existing compilers lack, but paramount to performance. Capitalizing on its seamless interoperability with Scala, we constructed a flexible frontend that leverages Scala's capabilities to add extensions. Moreover, we devised a streamlined operator interface that effortlessly facilitates a variety of evaluation strategies and index structures.

One of the main limitations of Flan currently is that we perform only very limited plan-level optimizations based on very simple heuristics. For instance, variable ordering of multi-way joins is currently derived from a left associative binary join plan based on the rule specification and uses hash indices by default. Users do have the option to specify their preferred join types and index types using global flags that applies these changes to all rule evaluations. However, much more sophisticated analysis and query planning could be done, for example, to choose between multi-way and binary joins, and for picking good variable orderings, or for picking tree-based or hash-based indices and so on at the granularity of each rule. For instance, as demonstrated in our evaluation in Section 6, the most effective evaluation strategy (e.g., multi-way or binary joins), often varies from one benchmark to another.

Flan achieves a commendable level of performance even in the absence of such clever query planning. We believe that Flan could be coupled with any existing well-studied query planning approach (e.g., using cardinality estimates to pick the index and join types, etc.) [Arch et al. 2022; Subotic et al. 2018] to improve performance further. Moreover, it is possible to add support for *dynamic* optimizations (e.g., adaptive join order optimization) using techniques such as on-stack replacement as demonstrated in prior work [Essertel et al. 2021].

Another promising line of future work is on integrating SMT constraints into Flan in the style of Formulog [Bembenek et al. 2020] that would unlock a wider range of static analysis including symbolic execution. GenSym [Wei et al. 2020, 2023], a state-of-the-art symbolic execution engine built using LMS, already equips LMS with functionalities to interface with SMT solvers. Consequently, much of the essential infrastructure is already in place. This integration would enable Flan to effectively function as a 'compiled Formulog' with minimal additional engineering effort. We plan to explore the feasibility and the effectiveness of this approach in our future work.

ACKNOWLEDGMENTS

We would like to thank our anonymous reviewers for their valuable feedback that helped improve the paper significantly. We thank Mikail Khan for his valuable contributions to the early iterations of the Flan implementation. We thank Guannan Wei, Oliver Bračevac, Patrick LaFontaine, and Pratyush Das for their comments on earlier drafts. This work was supported in part by NSF awards

1553471, 1564207, 1918483, 1910216, DOE award DE-SC0018050, as well as gifts from Meta, Google, Microsoft, and VMware.

REFERENCES

Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4 (2017), 20:1–20:44.

Supun Abeysinghe, Fei Wang, Grégory M. Essertel, and Tiark Rompf. 2023. Architecting Intermediate Layers for Efficient Composition of Data Management and Machine Learning Systems. *CoRR* abs/2311.02781 (2023).

Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley.

Nicholas Allen, Bernhard Scholz, and Padmanabhan Krishnan. 2015. Staged Points-to Analysis for Large Code Bases. In *CC (Lecture Notes in Computer Science, Vol. 9031)*. Springer, 131–150.

Nada Amin, William E. Byrd, and Tiark Rompf. 2019. Lightweight Functional Logic Meta-Programming. In *APLAS (Lecture Notes in Computer Science, Vol. 11893)*. Springer, 225–243.

Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph. D. Dissertation. Citeseer.

Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. 2017. Porting doop to Soufflé: a tale of inter-engine portability for Datalog-based analyses. In *SOAP@PLDI*. ACM, 25–30.

Samuel Arch, Xiaowen Hu, David Zhao, Pavle Subotic, and Bernhard Scholz. 2022. Building a Join Optimizer for Soufflé. In *LOPSTR (Lecture Notes in Computer Science, Vol. 13474)*. Springer, 83–102.

Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *SIGMOD Conference*. ACM, 1371–1382.

Michael Arntzenius and Neel Krishnaswami. 2020. Seminaïve evaluation for a higher-order functional language. *Proc. ACM Program. Lang.* 4, POPL (2020), 22:1–22:28.

Michael Arntzenius and Neelakantan R. Krishnaswami. 2016. Datafun: a functional Datalog. In *ICFP*. ACM, 214–227.

Albert Atserias, Martin Grohe, and Dániel Marx. 2008. Size Bounds and Query Plans for Relational Joins. In *FOCS*. IEEE Computer Society, 739–748.

Aaron Bembenek, Michael Greenberg, and Stephen Chong. 2020. Formulog: Datalog for SMT-based static analysis. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 141:1–141:31.

William C. Benton and Charles N. Fischer. 2007. Interactive, scalable, declarative program analysis: from prototype to implementation. In *PPDP*. ACM, 13–24.

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2014. Julia: A Fresh Approach to Numerical Computing. *CoRR* abs/1411.1607 (2014).

S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006a. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (Portland, OR, USA). ACM Press, New York, NY, USA, 169–190.

S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006b. *The DaCapo Benchmarks: Java Benchmarking Development and Analysis (Extended Version).* Technical Report TR-CS-06-01. http://www.dacapobench.org.

Ajay Brahmakshatriya and Saman P. Amarasinghe. 2021. BuildIt: A Type-Based Multi-stage Programming Framework for Code Generation in C++. In *CGO*. IEEE, 39–51.

Ajay Brahmakshatriya and Saman P. Amarasinghe. 2022. GraphIt to CUDA Compiler in 2021 LOC: A Case for High-Performance DSL Implementation via Staging with BuilDSL. In *CGO*. IEEE, 53–65.

Oliver Bračevac, Guannan Wei, Songlin Jia, Supun Abeysinghe, Yuxuan Jiang, Yuyan Bao, and Tiark Rompf. 2023. Graph IRs for Impure Higher-Order Languages: Making Aggressive Optimizations Affordable with Precise Effect Dependencies. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 236:1–236:31.

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*. ACM, 243–262.

Ashok K. Chandra and David Harel. 1985. Horn Clauses Queries and Generalizations. *J. Log. Program.* 2, 1 (1985), 1–15.

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *CADE (Lecture Notes in Computer Science, Vol. 4603)*. Springer, 183–198.

Grégory M. Essertel, Ruby Y. Tahboub, James M. Decker, Kevin J. Brown, Kunle Olukotun, and Tiark Rompf. 2018. Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data. In *OSDI*. USENIX Association, 799–815.

Grégory M. Essertel, Ruby Y. Tahboub, and Tiark Rompf. 2021. On-stack replacement for program generators and source-to-source compilers. In *GPCE*. ACM, 156–169.

Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh M. Patel. 2019. Scaling-up in-Memory Datalog Processing: Observations and Techniques. *Proc. VLDB Endow.* 12, 6 (Feb. 2019), 695–708.

Antonio Flores-Montoya and Eric M. Schulte. 2020. Datalog Disassembly. In *USENIX Security Symposium*. USENIX Association, 1075–1092.

Craig Freedman, Erik Ismert, and Per-Åke Larson. 2014. Compilation in the Microsoft SQL Server Hekaton Engine. *IEEE Data Eng. Bull.* 37, 1 (2014), 22–30.

Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting Worst-Case Optimal Joins in Relational Database Systems. *Proc. VLDB Endow.* 13, 11 (2020), 1891–1904.

Yoshihiko Futamura. 1971. Partial evaluation of computation process-an approach to a compiler-compiler. *Systems, Computers, Controls* 25 (1971), 45–50.

Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: thorough, declarative decompilation of smart contracts. In *ICSE*. IEEE / ACM, 1176–1186.

Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. 2006. *codeQuest:* Scalable Source Code Queries with Datalog. In *ECOOP (Lecture Notes in Computer Science, Vol. 4067)*. Springer, 2–27.

Julien Henry. 2022. User defined aggregate. https://github.com/souffle-lang/souffle/pull/2282/files.

Krystof Hoder, Nikolaj S. Bjørner, and Leonardo Mendonça de Moura. 2011. μZ- An Efficient Engine for Fixed Points with Constraints. In *CAV (Lecture Notes in Computer Science, Vol. 6806)*. Springer, 457–462.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation*. Prentice Hall.

Herbert Jordan, Pavle Subotic, David Zhao, and Bernhard Scholz. 2019a. Brie: A Specialized Trie for Concurrent Datalog. In *PMAM@PPoPP*. ACM, 31–40.

Herbert Jordan, Pavle Subotic, David Zhao, and Bernhard Scholz. 2019b. A specialized B-tree for concurrent datalog evaluation. In *PPoPP*. ACM, 327–339.

Herbert Jordan, Pavle Subotic, David Zhao, and Bernhard Scholz. 2022. Specializing parallel data structures for Datalog. *Concurr. Comput. Pract. Exp.* 34, 2 (2022).

Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce F. Duba. 1986. Hygienic Macro Expansion. In *LISP and Functional Programming*. ACM, 151–161.

Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. 2010. Generating code for holistic query evaluation. In *ICDE*. IEEE Computer Society, 613–624.

Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. 2005. Context-sensitive program analysis as database queries. In *PODS*. ACM, 1–12.

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA, 75–88.

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14.

Ondrej Lhoták and Kwok-Chiang Andrew Chung. 2011. Points-to analysis with efficient strong updates. In *POPL*. ACM, 3–16.

Boon Thau Loo, Tyson Condie, Minos N. Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2006. Declarative networking: language, execution and optimization. In *SIGMOD Conference*. ACM, 97–108.

Magnus Madsen, Jonathan Starup, and Ondrej Lhoták. 2022. Flix: A Meta Programming Language for Datalog. In *Datalog (CEUR Workshop Proceedings, Vol. 3203)*. CEUR-WS.org, 202–206.

Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. 2016. From Datalog to flix: a declarative language for fixed points on lattices. In *PLDI*. ACM, 194–208.

Frank McSherry, Michael Isard, and Derek Gordon Murray. 2015. Scalability! But at what COST?. In *HotOS*. USENIX Association.

Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *CIDR*. www.cidrdb.org.

Dan Moldovan, James M. Decker, Fei Wang, Andrew A. Johnson, Brian K. Lee, Zachary Nado, D. Sculley, Tiark Rompf, and Alexander B. Wiltschko. 2019. AutoGraph: Imperative-style Coding with Graph-based Performance. In *MLSys*. mlsys.org.

Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.

Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case Optimal Join Algorithms. *J. ACM* 65, 3 (2018), 16:1–16:40.

Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2013. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.* 42, 4 (2013), 5–16.

André Pacak and Sebastian Erdweg. 2022. Functional Programming with Datalog. In *ECOOP (LIPIcs, Vol. 222)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:28.

Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: A Data Platform Based on the Scaling-Up Approach. *Proc. VLDB Endow.* 11, 6 (2018), 663–676.

Rayon. 2022. Rayon: A data parallelism library for Rust. https://github.com/rayon-rs/rayon.

Thomas W. Reps. 1994. Solving Demand Versions of Interprocedural Analysis Problems. In *CC (Lecture Notes in Computer Science, Vol. 786)*. Springer, 389–403.

Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*. ACM Press, 49–61.

Tiark Rompf and Nada Amin. 2015. Functional pearl: a SQL to C compiler in 500 lines of code. In *ICFP*. ACM, 2–9.

Tiark Rompf and Nada Amin. 2019. A SQL to C compiler in 500 lines of code. *J. Funct. Program.* 29 (2019), e9.

Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. 2012. Scala-Virtualized: linguistic reuse for deep embeddings. *High. Order Symb. Comput.* 25, 1 (2012), 165–207.

Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*. ACM, 127–136.

Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. 2013. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. In *POPL*. ACM, 497–510.

Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. In *Datalog (CEUR Workshop Proceedings, Vol. 2368)*. CEUR-WS.org, 56–67.

Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. 1996. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theor. Comput. Sci.* 167, 1&2 (1996), 131–170.

Arash Sahebolamri, Langston Barrett, Scott Moore, and Kristopher Micinski. 2023. Bring Your Own Data Structures to Datalog. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 264:1–264:26.

Arash Sahebolamri, Thomas Gilray, and Kristopher K. Micinski. 2022. Seamless deductive inference via macros. In *CC*. ACM, 77–88.

Bernhard Scholz, Herbert Jordan, Pavle Subotic, and Till Westmann. 2016. On fast large-scale program analysis in Datalog. In *CC*. ACM, 196–206.

Bernhard Scholz, Kostyantyn Vorobyov, Padmanabhan Krishnan, and Till Westmann. 2015. A Datalog Source-to-Source Translator for Static Program Analysis: An Experience Report. In *ASWEC*. IEEE Computer Society, 28–37.

Jiwon Seo, Stephen Guo, and Monica S. Lam. 2015. SociaLite: An Efficient Graph Query Language Based on Datalog. *IEEE Trans. Knowl. Data Eng.* 27, 7 (2015), 1824–1837.

Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to Architect a Query Compiler. In *SIGMOD Conference*. ACM, 1907–1922.

Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big Data Analytics with Datalog Queries on Spark. In *SIGMOD Conference*. ACM, 1135–1149.

Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (2015), 1–69.

Yannis Smaragdakis and Martin Bravenboer. 2010. Using Datalog for Fast and Easy Program Analysis. In *Datalog (Lecture Notes in Computer Science, Vol. 6702)*. Springer, 245–251.

Pavle Subotic, Herbert Jordan, Lijun Chang, Alan D. Fekete, and Bernhard Scholz. 2018. Automatic Index Selection for Large-Scale Datalog Computation. *Proc. VLDB Endow.* 12, 2 (2018), 141–153.

Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing lattice-based program analyses in Datalog. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 139:1–139:29.

Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: a DSL for the definition of incremental program analyses. In *ASE*. ACM, 320–331.

Tamás Szabó, Markus Voelter, and Sebastian Erdweg. 2017. IncAL: A DSL for Incremental Program Analysis with Lattices. In *Proceedings of the International Workshop on Incremental Computing (IC)*.

Walid Taha and Tim Sheard. 1997. Multi-Stage Programming with Explicit Annotations. In *PEPM*. ACM, 203–217.

Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *SIGMOD Conference*. ACM, 307–322.

Luc Tielen. 2023. Eclair-lang. https://github.com/luc-tielen/eclair-lang.

Jeffrey D. Ullman. 1988. *Principles of Database and Knowledge-Base Systems, Volume I*. Principles of computer science series, Vol. 14. Computer Science Press.

Jeffrey D. Ullman. 1989. *Principles of Database and Knowledge-Base Systems, Volume II.* Computer Science Press. 984–987 pages.

Todd L. Veldhuizen. 2014. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *ICDT*. OpenProceedings.org, 96–106.

Jodat Vu. 2011. The art of multiprocessor programming by Maurice Herlihy and Nir Shavit. *ACM SIGSOFT Softw. Eng. Notes* 36, 5 (2011), 52–53.

Yisu Remy Wang, Max Willsey, and Dan Suciu. 2023. Free Join: Unifying Worst-Case Optimal and Traditional Joins. *Proc. ACM Manag. Data* 1, 2 (2023), 150:1–150:23.

Guannan Wei, Oliver Bracevac, Shangyin Tan, and Tiark Rompf. 2020. Compiling symbolic execution with staging and algebraic effects. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 164:1–164:33.

Guannan Wei, Yuxuan Chen, and Tiark Rompf. 2019. Staged abstract interpreters: fast and modular whole-program analysis via meta-programming. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 126:1–126:32.

Guannan Wei, Songlin Jia, Ruiqi Gao, Haotian Deng, Shangyin Tan, Oliver Bracevac, and Tiark Rompf. 2023. Compiling Parallel Symbolic Execution with Continuations. In *ICSE*. IEEE, 1316–1328.

Joel Wejdenstål. 2022. DashMap. https://github.com/xacrimon/dashmap.

John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. 2005. Using Datalog with Binary Decision Diagrams for Program Analysis. In *APLAS (Lecture Notes in Computer Science, Vol. 3780)*. Springer, 97–118.

John Whaley and Monica S. Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*. ACM, 131–144.

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29.

Eric Zhang. 2020. ekzhang/crepe: Datalog compiler embedded in Rust as a procedural macro. https://github.com/ekzhang/crepe.

Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. *CoRR* abs/2304.04332 (2023).

Yihong Zhang, Yisu Remy Wang, Max Willsey, and Zachary Tatlock. 2022. Relational e-matching. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–22.

David Zhao, Pavle Subotic, Mukund Raghothaman, and Bernhard Scholz. 2021. Towards Elastic Incrementalization for Datalog. In *PPDP*. ACM, 20:1–20:16.

David Zhao, Pavle Subotic, and Bernhard Scholz. 2020. Debugging Large-scale Datalog: A Scalable Provenance Evaluation Strategy. *ACM Trans. Program. Lang. Syst.* 42, 2 (2020), 7:1–7:35.

Philip Zucker. 2022. Logging an Egg: Datalog on E-Graphs *(EGRAPHS 2022)*. 1–6.

Received 2023-07-11; accepted 2023-11-07