Bring Your Own Data Structures to Datalog

ARASH SAHEBOLAMRI, Syracuse University, USA
LANGSTON BARRETT, Galois, USA
SCOTT MOORE, Galois, USA
KRISTOPHER MICINSKI, Syracuse University, USA

The restricted logic programming language Datalog has become a popular implementation target for deductive-analytic workloads including social-media analytics and program analysis. Modern Datalog engines compile Datalog rules to joins over explicit representations of relations—often B-trees or hash maps. While these modern engines have enabled high scalability in many application domains, they have a crucial weakness: achieving the desired algorithmic complexity may be impossible due to representation-imposed overhead of the engine's data structures. In this paper, we present the "Bring Your Own Data Structures" (Byods) approach, in the form of a DSL embedded in Rust. Using Byods, an engineer writes logical rules which are implicitly parametric on the concrete data structure representation; our implementation provides an interface to enable "bringing their own" data structures to represent relations, which harmoniously interact with code generated by our compiler (implemented as Rust procedural macros). We formalize the semantics of Byods as an extension of Datalog's; our formalization captures the key properties demanded of data structures compatible with Byods, including properties required for incrementalized (semi-naïve) evaluation. We detail many applications of the Byods approach, implementing analyses requiring specialized data structures for transitive and equivalence relations to scale, including an optimized version of the Rust borrow checker Polonius; highly-parallel PageRank made possible by lattices; and a large-scale analysis of LLVM utilizing index-sharing to scale. Our results show that Byods offers both improved algorithmic scalability (reduced time and/or space complexity) and runtimes competitive with state-of-the-art parallelizing Datalog solvers.

CCS Concepts: • **Software and its engineering → Domain specific languages**; • **Theory of computation → Constraint and logic programming**.

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

ACM Reference Format:
Arash Sahebolamri, Langston Barrett, Scott Moore, and Kristopher Micinski. 2023. Bring Your Own Data Structures to Datalog. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 264 (October 2023), 26 pages. https://doi.org/10.1145/3622840

Authors' addresses: Arash Sahebolamri, asahebol@syr.edu, Syracuse University, Syracuse, USA; Langston Barrett, langston@galois.com, Galois, Portland, USA; Scott Moore, scott@galois.com, Galois, Portland, USA; Kristopher Micinski, kkmicins@syr.edu, Syracuse University, Syracuse, USA.

2475-1421/2023/10-ART264
https://doi.org/10.1145/3622840

1 INTRODUCTION

Declarative programming languages are an attractive candidate for the implementation of systems based on formal rules, including industrial-scale program analyses [Bravenboer and Smaragdakis 2009; Smaragdakis and Bravenboer 2011], type systems [Pacak et al. 2020], network analytics [Lopes et al. 2016] and graph algorithms [Wang et al. 2018]. These languages balance expressivity and implementation strategy, promising their users an optimal implementation substrate tailored to their application domain. For example, Soufflé compiles Horn clauses to iterated joins [Jordan et al. 2016]; Datafun generalizes these rules to programming with monotonic maps [Arntzenius and Krishnaswami 2019, 2016], and Formulog further allows incorporating satisfiability-modulo theories (SMT) [Bembenek et al. 2020].

Datalog, *i.e.*, first-order Horn clauses, serves as a common basis for many modern declarative languages tailored towards the high-performance operationalization of deductive-analytic workloads. Horn clauses naturally enable chain-forward programming, which may be operationalized in a bottom-up fashion (*e.g.*, via semi-naïve evaluation [Bancilhon and Ramakrishnan 1986]). Datalog's restriction to first-order Horn clauses has enabled orders-of-magnitude scalability gains in declaratively-specified program analysis (e.g., DOOP [Smaragdakis and Bravenboer 2011]) due to the efficiency of iterated joins over concurrent B-trees and tries [Jordan et al. 2019].

Unfortunately, users of Datalog face a crucial challenge—the data structures exposed by the underlying engine become a leaky abstraction, and induce representation-imposed blowup in workloads ill-suited to the *sets-of-tuples* representation. For example, Datalog-based implementations of tasks which rely upon union-find (common in type synthesis and compiler optimization) face intrinsic blowup due to their need to explicitly materialize all facts of an equivalence relation (see Table 1 in Section 5.1). One modern engine combats this challenge in an ad-hoc fashion, allowing the user to chose among a finite number of engine-provided data structures on a per-relation basis (e.g., the eqrel relations in Soufflé). While this partially remediates the issue in case the user happens to need exactly the data structures provided by the engine, the user is out of luck when the demands of their application domain have not been foreseen by engine authors.

There is another angle from which we can view the issue. While marking a relation as an equivalence changes the semantics of a Datalog program, equivalence relations do not extend the expressive power of Datalog in principle, as one could include auxiliary rules necessary to materialize the equivalence. Many extensions of Datalog (e.g., Datalog^{FS} [Mazuran et al. 2013], Flix [Madsen et al. 2016], and DeALS [Shkapsky et al. 2015]) seek to extend the expressive power of Datalog, while retaining efficient evaluation strategies like semi-naïve evaluation. For example, Flix extends Datalog with non-powerset lattices, necessary for declarative implementation of programs such as constant propagation analysis and shortest-path computations. One may wonder if a new Datalog variant is required for each such application.

In this work we ask: how can we harmoniously integrate user-provided data structures with Horn clauses, so that a user may write logical rules parametric to the underlying data structure representation? We propose the "Bring Your Own Data Structures" approach. In this approach, the user supplies a data structure which exposes an interface for fact enumeration; Horn clauses are then compiled to joins which perform tuple enumeration indirectly, via a *concretization function.* To enable users to bring robust, high-performance data structures to Datalog, our tool, Byods[1], is implemented as an embedded domain-specific language (EDSL) within Rust [Matsakis and Klock 2014] via procedural macros. In contrast to approaches which require users to pick from a set of engine-provided relations (such as DER [Jordan et al. 2022]), users are free to build space-and-time-efficient relation-backing data structures, achieving state-of-the-art efficiency using all of

[1] Rhymes with "roads."

the features available in Rust. Our macro-based compiler (detailed in Section 4) emits code which interacts with these user-provided data structures via a protocol inspired by our formalization.

To rigorously define the semantics of Byods we present a core formalism, DL_{DS}, which explicates the interface demanded of user-provided data structures. Our formalism, presented in section 3, builds upon the intuition that user-provided relations are "standing in for" extensionally-manifest relations: custom relations, reminiscent of galois connections, are formalized as lattices equipped with the aforementioned concretization function, which allows extensional enumeration—we then extend Datalog's fixed point-based semantics to work in terms of this concretization operator rather than a "hard wired" representation, recovering Datalog's typical interpretation by taking the lattice to be the power set of facts. We extend our formalism to account for semi-naïve evaluation and detail its ramifications for the design of data structures users bring themselves. We then scale DL_{DS} to a complete implementation as a Rust EDSL, detailing several semantics-preserving optimizations necessary to achieve an implementation competitive with state-of-the-art parallel engines.

To evaluate the utility and flexibility of the Byods approach, we have implemented a wide variety of specialized data structures, each meant to excel in a given application domain. In section 5, we present several evaluations, including Steensgaard analysis using union-find, an optimized implementation of the Rust borrow checker, index sharing using bipartite graph matching, parallel PageRank, and a large-scale context-sensitive analysis of LLVM IR (which scales to Redis and SQLite). Our applications show that Byods offers an ideal platform for implementing highly-scalable deductive-analytic workloads without the compromises imposed by engine-supplied tuple representations.

Specifically, this paper offers the following contributions:

- Byods, a framework for customizing relation-backing data structures and extending Datalog semantics through specialized data structures.
- DL_{DS}, a formal language extending Datalog with abstract data structures, including its fixed point and incrementalized semantics, and properties that abstract data structures must satisfy for equivalence of the two.
- An evaluation showing several customized data structures implemented in this framework, and how each one provides performance benefits or expressivity benefits compared to Datalog. The data structures include specialized equivalence and transitive relations, index-sharing for relations, and user-defined lattices.

2 OVERVIEW

As Datalog is used to solve problems in more domains, Datalog users find themselves having to live with the choices made for them by Datalog engine authors, in particular, the choices of data structures that back relations. From Binary Decision Diagrams [Whaley et al. 2005], to B-trees and hash maps, each data structure has its strengths and weaknesses when it comes to storing relation data. B-trees allow efficient index-sharing [Subotić et al. 2018], but may be slower than hash maps and require values to be totally ordered. Hash maps usually perform better when it comes to raw performance compared to B-trees, but do not admit straightforward index-sharing strategies. Binary Decision Diagrams allow dense representation of facts, but require the values they store to be essentially small bit-sequences. In addition, modern Datalog engines provide parallelism, which requires concurrent data structures. Designing concurrent data structures is also not a once-and-for-all solved problem, different strategies exist with varying tradeoffs.

For certain tasks, special-purpose data structures are *necessary* to achieve the desired algorithmic complexity. For example, equivalence relations [Nappa et al. 2019] perform significantly better in time and space complexity with special-purpose data structures compared to a naïve implementation backed by a B-tree-based tuple representation. Many general-purpose programming languages permit their users to define new data structures to satisfy their use cases, and often have a growing ecosystem of high quality data structures that their users can choose from. Datalog has not been able to afford its users the same freedom, as it is a declarative, logic programming language, not one suited for modern data structure implementation. In this paper, we present Byods, a solution to this problem. We realize Byods as a Datalog engine embedded in Rust in the style of Ascent [Sahebolamri et al. 2022] and Crepe [Zhang 2023], one that extends Datalog to allows users to implement their own data structures to back relations in their Datalog programs. This approach has three main benefits:

- The user is no longer bound by the choice of data structures made for them by the authors of the Datalog engine. If the in-box data structures are not well-optimized, the user can implement their own data structures.
- It is possible to utilize or define new special-purpose data structures, such as union-find data structures, offering significant run time speedups.
- The Datalog user can extend the semantics of Datalog by defining data structures whose semantics cannot be emulated by Datalog rules, gaining the ability to express more logical programs in Datalog. For example, programs that require lattices or recursive aggregation are expressible in Byods (5.6).

Byods defines a macro-based protocol for the Datalog compiler to interact with data structure providers. This macro based protocol allows the Datalog engine to provide information about a relation's use in the Datalog program to the data structure provider, and allows data structure providers to employ arbitrary logic in choosing data structures.

We use a simple example to show how Byods works. Graph mining and network analysis tasks are major applications of Datalog [Seo et al. 2013]. A common and routine task in graph mining is computing the transitive closure of the input graph. Scrutinizing graphs of social networks, we realize that they usually are made up of relatively large communities, where each node in the community has a path to every other node. Storing all the connected node pairs of a community (which would be K^2 facts for a community of size K and is what a Datalog program would do) is suboptimal. We can instead implement a data structure for transitive relations, `TrRelUnionFind<T>`, that is optimized to handle graphs/relations made up of large communities. We can then plug it into our Datalog program, speeding up our Datalog computation without sacrificing the benefits of using a declarative language (as we do in 5.4).

```
// program with explicit rules for
// transitive closure

relation path(Node, Node)
path(x, y) :- edge(x, y).
path(x, z) :- path(x, y), edge(y, z).
// ...
```

```
// program with a customized data
// structure for transitive relations
#[ds(trrel_uf)]
relation path(Node, Node)
path(x, y) :- edge(x, y).

// ...
```

In the code snippet above, on the right, we have removed the explicit rule for making `path` the transitive closure of `edge`, and have instead tagged `path` with `trrel_uf`, the data structure provider for transitive relations that we have defined. As we'll discuss in more detail in Section 4, a data structure provider is a Rust module that implements a number of macros for various components of a relation. For example, `rel_ind_common!` is the macro that evaluates to the type implementing the data structure shared by all indices of a relation (the Datalog engine interacts with a relation through its indices). In our example, this macro invocation evaluates to (a type containing) our data structure for transitive relations: `TrRelUnionFind<Node>`.

We should emphasize that the point of Byods is not any single data structure provider. Rather, it is extensibility: the fact that Byods makes it possible for the Datalog user to define new data structure providers and plug them into their Datalog computations.

3 DATALOG WITH CUSTOM RELATIONS

This section introduces $\mathrm{DL}_{\mathrm{DS}}$, a language that captures the essence of Byods by generalizing Datalog to enable custom relations. We'll introduce both a fixed point semantics, and an incrementalized semantics for $\mathrm{DL}_{\mathrm{DS}}$, and present properties that abstract data structures must satisfy for the equivalence of the two.

Custom relations in $\mathrm{DL}_{\mathrm{DS}}$ are defined by a triple $(D,\ inj,\ \boldsymbol{\gamma})$. For a relation $\mathtt{rel}(T)$ tagged by such an abstraction, D is an abstract domain representing the modified behavior/characteristics of the relation. We require D to be a lattice and therefore equipped with an idempotent, associative, and commutative join operator. $inj : T \rightarrow D$ is the *injection function*, and $\boldsymbol{\gamma} : D \rightarrow \mathcal{P}(T)$ is the *concretization function*. These abstractions encapsulate the interactions between a custom relation-backing data structure and the Datalog engine per-se: the Datalog engine stores instances of D, updates them by asking new tuples to be added to stored instances of D (analogous to updating an instance as follows: $d' = d \sqcup inj(t)$ where t is the new tuple to be added), and enumerates the tuples in a stored instance by calling the concretization function $\boldsymbol{\gamma}$.

We assume every relation `rel` in a $\mathrm{DL}_{\mathrm{DS}}$ program is equipped with an abstract data structure $(D_{\mathtt{rel}},\ inj_{\mathtt{rel}},\ \boldsymbol{\gamma}_{\mathtt{rel}})$. For relations without a custom backing data structure (i.e., normal Datalog relations), the abstract domain is the powerset of the type of tuples of the relation ($\mathcal{P}(T)$), and the injection and concretization functions are simply the singleton-set creation and identity functions. In case there are no custom relations in a $\mathrm{DL}_{\mathrm{DS}}$ program, the program is equivalent to its Datalog counterpart.

$\mathrm{DL}_{\mathrm{DS}}$'s syntax is identical to Datalog: a rule in $\mathrm{DL}_{\mathrm{DS}}$ has a head atom and a set of body atoms, where an atom is a relation symbol followed by a list of arguments (variables `Var` or constant symbols `Val`). A $\mathrm{DL}_{\mathrm{DS}}$ program is a collection of rules.

Following the Datalog tradition [Ceri et al. 1989], we define a fixed point semantics for $\mathrm{DL}_{\mathrm{DS}}$. The key component of the semantics is the *immediate consequence operator*. This operator for a rule R in *Datalog* is a function $T_R : \mathrm{DB} \rightarrow \mathrm{DB}$ that adds to the input database db all the facts derivable in one step when R is treated as an inference rule and the facts in db are treated as axioms. In Datalog, a $\mathrm{db} \in \mathrm{DB}$ is a set of facts, where each fact is a relation symbol followed by a sequence of constant symbols.

For $\mathrm{DL}_{\mathrm{DS}}$ we need to modify this operator and the definition of a database to accommodate custom relations. In $\mathrm{DL}_{\mathrm{DS}}$, a database is a tuple of abstract data structures, one for each relation in the program. Since abstract data structures for relations (Ds) are lattices, a database is also a lattice with product ordering. The notation db @ `r` selects the abstract data structure for relation `r` from database db. The immediate consequence operator for a rule R in $\mathrm{DL}_{\mathrm{DS}}$ is defined as follows.

$$T_R(\mathrm{db}) = \sqcup\big\{ inj_{\,\mathrm{headrel}(R)}\big(\mathrm{head}(R)[\theta]\big) \;\big|\; \theta : \mathtt{Var} \rightharpoonup \mathtt{Val}.\; \forall \mathtt{r}(xs) \in \mathrm{body}(R).\; xs[\theta] \in \boldsymbol{\gamma}_{\mathtt{r}}(\mathrm{db} \;@\; \mathtt{r}) \big\}$$

In words, the immediate consequence of a rule finds substitutions (θs) of values for variables that make all the body atoms of the rule present in their respective concretizations in the current database, and builds up a lattice by injecting the corresponding head tuples into the abstract domain of the head relation.

This definition is similar to the definition of the immediate consequence operator for Datalog, except injection and concretization function calls are inserted where required to interact with custom data structures.

The immediate consequence operator for a program lifts this definition to a collection of rules. We define an updated immediate consequence operator for rules $T'_R(\text{db})$ where $T'_R(\text{db})\ @\ \text{headrel}(R) = T_R(\text{db})$, and for every other relation $\mathtt{r}$, $T'_R(\text{db})\ @\ \mathtt{r} = \bot$. The sole purpose of this updated definition is for the output of the function to match the schema of DL_{DS} databases. This definition also hints at another requirement for abstract data structures: they need to have a bottom element. With this updated definition, the immediate consequence operator for a program P is defined as:

$$T_P(\text{db}) = \text{db} \sqcup \Big(\bigsqcup_{R \in P} T'_R(\text{db}) \Big)$$

The fixed point semantics of DL_{DS} is the least fixed point of T_P. Note that for the semantics to be well defined, T_P needs to be a monotonic function [Tarski 1955]. Monotonicity is defined in the usual way: a function T_P is monotonic in case for all db1 and db2, db1 $\sqsubseteq$ db2 implies $T_P(\text{db1}) \sqsubseteq T_P(\text{db2})$. In Datalog T_P is guaranteed to be monotonic, but in DL_{DS}, monotonicity of T_P is contingent on the behavior of abstract data structures. We specify sufficient conditions for monotonicity of T_P:

Theorem 3.1. *If all the concretization functions ($\boldsymbol{\gamma}$s) associated with abstract data structures are monotonic, T_P is monotonic and the program P has a well defined semantics.*

Proof. To prove this theorem, we start by showing that monotonicity of $\boldsymbol{\gamma}$s implies that T_Rs are monotonic. We assume db1 $\sqsubseteq$ db2, from the monotonicity of $\boldsymbol{\gamma}$s, we know that for all body relations $\mathtt{r}$ in a rule R, $\boldsymbol{\gamma}(\text{db1@}\mathtt{r}) \subseteq \boldsymbol{\gamma}(\text{db2@}\mathtt{r})$, therefore for all substitution schemes θ and all body atoms $\mathtt{r}(xs)$ of R, if $\mathtt{r}(xs)[\theta] \in \boldsymbol{\gamma}(\text{db1@}\mathtt{r})$ then $\mathtt{r}(xs)[\theta] \in \boldsymbol{\gamma}(\text{db2@}\mathtt{r})$. This implies that the set of tuples fed to the injection function for $T_R(\text{db1})$ is a subset of the set of tuples fed to the injection function for $T_R(\text{db2})$. Since both sets go through the same projection (the injection function for the rule head), then the post-injection set of elements for $T_R(\text{db1})$ is also a subset of the post-injection set of elements for $T_R(\text{db2})$; from which we conclude that $T_R(\text{db1}) \sqsubseteq T_R(\text{db2})$. Monotonicity of T_P directly follows from monotonicity of T_Rs. □

This theorem gives us a grip on what kinds of abstract data structures are well-behaved in Byods and what kinds are problematic. For example we expect `eqrel`, an abstract data structure for equivalence relations (5.1) to be well-behaved and indeed, it is. The DL_{DS} version of `eqrel` can be defined as:

$$\begin{aligned} D_{\text{eqrel}} &= \mathcal{P}(T) \\ inj_{\text{eqrel}}(t) &= \{t\} \\ \boldsymbol{\gamma}_{\text{eqrel}}(s) &= (\text{equivalence closure of } s) \end{aligned}$$

In this definition, $\boldsymbol{\gamma}_{\text{eqrel}}$ is clearly monotonic, satisfying the requirements for well-behaved programs. From Theorem 3.1 we can conclude other kinds of custom relations are also well-behaved: relations that have built-in filtering or projection are other examples of well-behaved relations. The theorem also gives us an idea of what custom relation data structures may not be well-behaved.

For example, lattices á la Flix [Madsen et al. 2016] can be implemented using our custom relations approach. However, Theorem 3.1 does not provide a guarantee that they would be well-behaved. Whether a program with Flix-style lattices is well-behaved depends on the structure of the rules, and no blanket guarantee exists for its well-behavedness.

Composition through custom domains. An interesting aspect of $\mathrm{DL}_{\mathrm{DS}}$ is that it allows composition through custom domains. That is, an abstract domain D for a relation can itself be defined via a Datalog (or $\mathrm{DL}_{\mathrm{DS}}$) program! Take the eqrel example. One can define eqrel through a Datalog program P_{eq} with relations $\mathtt{req}_{\mathtt{in}}$ and $\mathtt{req}_{\mathtt{out}}$ and rules ensuring $\mathtt{req}_{\mathtt{out}}$ is the equivalence closure of $\mathtt{req}_{\mathtt{in}}$. The injection function associated with eqrel in that case would be the fixed point of $T_{P_{\mathrm{eq}}}$ reached from the database containing a single fact: the input tuple as a fact of the relation $\mathtt{req}_{\mathtt{in}}$. The concretization function would return the tuples of $\mathtt{req}_{\mathtt{out}}$ in the output database of evaluation of P_{eq}. Finally, the abstract domain associated with this $\mathrm{DL}_{\mathrm{DS}}$-defined custom relation would be the fixed points of $T_{P_{\mathrm{eq}}}$, which, by Tarski's fixed point theorem [Tarski 1955], form a lattice.

$$\begin{aligned} D_{\mathrm{eqrel}} &= \{\mathrm{db} \mid \mathrm{db} = T_{P_{\mathrm{eq}}}(\mathrm{db})\} \\ \mathit{inj}_{\mathrm{eqrel}}(t) &= T^{*}_{P_{\mathrm{eq}}}(\{\mathtt{req}_{\mathtt{in}}(t)\}) \\ \boldsymbol{\gamma}_{\mathrm{eqrel}}(\mathrm{db}) &= \{t \mid \mathtt{req}_{\mathtt{out}}(t) \in \mathrm{db}\} \end{aligned}$$

One can show that this style of composition yields $\mathrm{DL}_{\mathrm{DS}}$ programs that behave the same as their explicitly inlined versions. That is, a $\mathrm{DL}_{\mathrm{DS}}$ program P with relation $\mathtt{r}$ tagged with a custom domain D that is itself defined by another $\mathrm{DL}_{\mathrm{DS}}$ program P_D with normal designated relations $\mathtt{r}_{\mathtt{in}}$ and $\mathtt{r}_{\mathtt{out}}$ is equivalent to its inlined version P^E; where $\mathtt{r}$ is removed, appearances of $\mathtt{r}$ in rule heads and bodies in P are replaced by $\mathtt{r}_{\mathtt{in}}$ and $\mathtt{r}_{\mathtt{out}}$ respectively, and rules of P_D are included in P^E (assuming there are no common relation names between P and P_D). Existence of this composition strategy for $\mathrm{DL}_{\mathrm{DS}}$ provides a potential for making $\mathrm{DL}_{\mathrm{DS}}$ programs modular.

The fixed point semantics of $\mathrm{DL}_{\mathrm{DS}}$ provides an evaluation strategy: starting with the empty (bottom) database and applying the immediate consequence operator successively, until a fixed point is reached. The inefficiency of this evaluation strategy forces us to study the implications of incrementalized evaluation of $\mathrm{DL}_{\mathrm{DS}}$ programs.

3.1 Incrementalized Semantics

The semantics presented above is analogous to the *naïve* evaluation strategy of Datalog, where the same ground version [2] of a rule fires over and over again over the course of evaluation (it fires in every iteration after the first time it fires until a fixed point is reached). To avoid this useless work, Datalog engines employ an incrementalized[3] evaluation strategy called *semi-naïve* evaluation [Bancilhon 1986]. To study the implications of semi-naïve evaluation for $\mathrm{DL}_{\mathrm{DS}}$, we present the semi-naïve semantics of $\mathrm{DL}_{\mathrm{DS}}$. To ease our way into semi-naïve evaluation of $\mathrm{DL}_{\mathrm{DS}}$ and its ramifications for custom relations, we start by presenting a semi-naïve semantics for Datalog and proving its equivalence to the naïve evaluation strategy of Datalog.

We employ a slightly modified version of T_P that records at what iteration each fact was added to the database. We do this by providing the iteration number as an argument:

[2]A ground version of a rule is a rule with all its variables substituted with concrete values. It's trivially the case that in Datalog and $\mathrm{DL}_{\mathrm{DS}}$, a rule can be replaced by all its ground versions.

[3] Not to be confused with incremental Datalog solvers (e.g., DDlog [Ryzhyk and Budiu 2019] and the work of [Szabó et al. 2021]). An incremental Datalog solver is able to speedup reevaluating a Datalog program when the input set of facts changes by reusing the previous evaluation.

$T_P^i(\mathrm{db}) = \mathrm{db} \cup (T_P(\mathrm{db}) - \mathrm{db})^i$

That is, every new fact at iteration i is tagged with i. We now define the semi-naïve semantics of Datalog. In this semantics, each rule $h \leftarrow b_1, b_2, ..., b_n$ is duplicated $2^n - 1$ times [4], where body atoms acquire a *version* tag: τ (for total) or Δ. All the combinations of τ and Δ are present except the all-τ combination. The semi-naïve immediate consequence operator T_P^{SN} operates over pairs of databases, one is the total database, from which τ atoms read, and the other is the delta database, from which the Δ atoms read. The intuition is that to produce a new fact at iteration i, a rule must examine at least one fact produced in iteration $i-1$; these are exactly the facts stored in the delta database. Iteration in the semi-naïve semantics of Datalog happens as follows: [5]

$$
\begin{aligned}
\mathrm{db}_\tau^{i+1} &= \mathrm{db}_\tau^i \cup \mathrm{db}_\Delta^i \\
\mathrm{db}_\Delta^{i+1} &= T_P^{SN}(\mathrm{db}_\tau^i,\ \mathrm{db}_\Delta^i) - (\mathrm{db}_\tau^i \cup \mathrm{db}_\Delta^i)
\end{aligned}
$$

To show that the naïve and semi-naïve semantics of Datalog are equivalent, we present the following lemma.

Lemma 3.2. *In Datalog, each iteration of T_P^{SN} discovers the same new facts as the same iteration of T_P.*

Proof. We prove the lemma by contradiction. Assume iteration i is the first iteration where there are new facts discovered by T_P and not T_P^{SN} (it's straightforward to rule out the reverse case, since $\mathrm{db}^i = \mathrm{db}_\tau^i \cup \mathrm{db}_\Delta^i$, every rule in T_P^{SN} operates over subsets of facts from the corresponding rule in T_P). The facts newly discovered by T_P and not T_P^{SN} can only originate from the absent all-τ version of a rule. Assume there is a new fact that would be discovered by the all-τ version of a rule. From the facts that match the body of the rule, causing it to fire, let j be the iteration of the newest fact(s), j must be smaller than $i-1$ (otherwise an all-τ rule would not fire). At iteration $j+1$, there would have been a version of the rule containing Δs that would have fired (since $j+1$ is the first iteration in which the facts appear in the input database, they must be in the delta database at iteration $j+1$), discovering the fact. Since $j+1 < i$, we have reached a contradiction, which completes the proof. □

To incrementalize $\mathrm{DL}_{\mathrm{DS}}$, we follow a similar strategy. But instead of having τ atoms read from the total database and Δ atoms read from the delta database through the $\boldsymbol{\gamma}$ concretization function, we require abstract data structures to have a pair of concretization functions: $\boldsymbol{\gamma}_\tau$ and $\boldsymbol{\gamma}_\Delta$, for τ atoms and Δ atoms respectively. These two functions take the pair of total and delta databases as input (and return a set of tuples like $\boldsymbol{\gamma}$). Furthermore, we require the following properties of $\boldsymbol{\gamma}_\tau$ and $\boldsymbol{\gamma}_\Delta$:

$$\boldsymbol{\gamma}_\tau(\mathrm{db}_\tau, \mathrm{db}_\Delta) \cup \boldsymbol{\gamma}_\Delta(\mathrm{db}_\tau, \mathrm{db}_\Delta) = \boldsymbol{\gamma}(\mathrm{db}_\tau \sqcup \mathrm{db}_\Delta) \tag{1}$$

$$\boldsymbol{\gamma}_\Delta(\mathrm{db}_\tau, \mathrm{db}_\Delta) \supseteq \boldsymbol{\gamma}(\mathrm{db}_\tau \sqcup \mathrm{db}_\Delta) - \boldsymbol{\gamma}(\mathrm{db}_\tau) \tag{2}$$

Property (1) is self explanatory. The intuition for Property (2) (and introduction of $\boldsymbol{\gamma}_\tau$ and $\boldsymbol{\gamma}_\Delta$ in the first place) is to ensure facts don't skip the delta database. We'll discuss this more at the end of this section.

[4] If $n = 0$, we leave the rule alone!

[5] We'll omit the iteration superscript on T_P for the rest of this section, as it will clutter the presentation and will be a distraction. We however will make frequent use of the notion of the iteration at which a fact was discovered.

For DL_DS, we also simplify the iteration scheme by not deduplicating facts across db_τ and db_Δ, noting (without proof) that a version of the semi-naïve semantics with deduplication can be constructed that is equivalent to our semantics.

$$\begin{aligned}\text{db}_\tau^{i+1} &= \text{db}_\tau^i \sqcup \text{db}_\Delta^i \\ \text{db}_\Delta^{i+1} &= T_P^{SN}(\text{db}_\tau^i, \text{db}_\Delta^i)\end{aligned}$$

We prove the equivalence of semi-naïve and naïve semantics for DL_DS in two steps. Before taking the first step, we introduce yet another semantics: the super-duper-naïve semantics of DL_DS! Super-duper naïve semantics is just like the semi-naïve semantics, except it also includes all-τ versions of rules, defeating the purpose of semi-naïve semantics and hence the name. In the first step, we prove the equivalence of the super-duper-naïve and semi-naïve semantics. In the second step, we prove the equivalence of the super-duper-naïve and the naïve semantics.

LEMMA 3.3. *For a DL_{DS} program P, T_P^{SDN} (from super-duper-naïve semantics) and T_P^{SN} produce the same databases at each iteration.*

PROOF. We prove the lemma by contradiction. Assume iteration i is the first iteration in which the outputs of T_P^{SDN} and T_P^{SN} diverge. From the definitions, it follows that there is an all-τ version of a ground rule R in T_P^{SDN} that has fired for the first time, causing the divergence. Let j be the iteration at which the newest fact f matching the body of the rule appeared in either $\boldsymbol{\gamma}_\tau$ or $\boldsymbol{\gamma}_\Delta$.

From properties (1) and (2), we know $\boldsymbol{\gamma}_\Delta^j \cup \boldsymbol{\gamma}_\tau^j = \boldsymbol{\gamma}^j$, and $\boldsymbol{\gamma}_\Delta^j \supseteq \boldsymbol{\gamma}^j - \boldsymbol{\gamma}(\text{db}_\tau^j)$ [6]. From the iteration scheme we have $\text{db}_\tau^j = \text{db}_\tau^{j-1} \sqcup \text{db}_\Delta^{j-1}$, from which we get $\boldsymbol{\gamma}(\text{db}_\tau^j) = \boldsymbol{\gamma}^{j-1}$. Put together, we conclude that $\boldsymbol{\gamma}_\Delta^j \supseteq \boldsymbol{\gamma}^j - \boldsymbol{\gamma}^{j-1}$. Since j is the first iteration at which the fact appeared in $\boldsymbol{\gamma}$, it means it was not in $\boldsymbol{\gamma}^{j-1}$, therefore it was in $\boldsymbol{\gamma}_\Delta^j$.

f appearing in $\boldsymbol{\gamma}_\Delta^j$ means a version of the rule R containing Δs fired at iteration j for both T_P^{SDN} and T_P^{SN}, and since $j \le i$, it rendered subsequent firings of the rule without effect (given monotonicity of $\boldsymbol{\gamma}$s and the iteration scheme), preventing divergence of T_P^{SDN} and T_P^{SN} at iteration i. This contradicts our assumption and completes the proof. □

LEMMA 3.4. *For a DL_{DS} program P, super-duper-naïve and naïve semantics are equivalent.*

PROOF. We show that for any decomposition of a database db into db_τ and db_Δ such that $\text{db} = \text{db}_\tau \sqcup \text{db}_\Delta$, $T_P^{\text{SDN}}(\text{db}_\tau, \text{db}_\Delta) = T_P(\text{db})$; from which the lemma follows.

Take a ground version of a rule that fires in T_P, for all its body facts $\mathrm{r}(t)$, t is in the concretization of r in the database (i.e., $t \in \boldsymbol{\gamma}(\text{db} \mathbin{@} \mathrm{r})$). From property (1) of $\boldsymbol{\gamma}_\tau$ and $\boldsymbol{\gamma}_\Delta$, it follows that t is either in $\boldsymbol{\gamma}_\tau$ or $\boldsymbol{\gamma}_\Delta$. Since all the τ and Δ combinations of the rule are present in T_P^{SDN}, there is at least one version of the rule in T_P^{SDN} that fires, causing the same tuple to be injected into the output of T_P^{SDN}. Conversely, since all the rules in T_P^{SDN} read from databases that are subsumed by db, if a ground version of a rule fires in T_P^{SDN}, the corresponding rule also fires in T_P, again causing the same tuple to be injected into the output of T_P. □

THEOREM 3.5. *The semi-naïve and naïve semantics of DL_{DS} are equivalent.*

PROOF. Using Lemmas 3.3 and 3.4. □

Having presented the semi-naïve semantics for DL_DS and properties required of $\boldsymbol{\gamma}_\tau$ and $\boldsymbol{\gamma}_\Delta$, we can reflect on them now. The first thing to note is that the simplistic approach of setting $\boldsymbol{\gamma}_\tau$ =

[6]We define $\boldsymbol{\gamma}_{\tau,\Delta}^j = \boldsymbol{\gamma}_{\tau,\Delta}(\text{db}_\tau^j,\ \text{db}_\Delta^j)$; and $\boldsymbol{\gamma}^j = \boldsymbol{\gamma}(\text{db}_\tau^j \sqcup \text{db}_\Delta^j)$.

$\boldsymbol{\gamma}(\mathrm{db}_\tau)$ and $\boldsymbol{\gamma}_\Delta = \boldsymbol{\gamma}(\mathrm{db}_\Delta)$ works for normal (semantics-preserving) relations, but it will not work in general. Take eqrel that we introduced earlier for example, if we follow the above simplistic approach for eqrel, there could be facts that would show up in $\boldsymbol{\gamma}_\tau$ without showing up in $\boldsymbol{\gamma}_\Delta$ first. For example, if for an eqrel, at some iteration $\mathrm{db}_\tau = \{(1, 2)\}$ and $\mathrm{db}_\Delta = \{(2,\ 3)\}$, the fact $(1,\ 3)$ would not be in $\boldsymbol{\gamma}_\Delta$, it would only show up in $\boldsymbol{\gamma}_\tau$ in the next iteration. This would undermine the semi-naïve evaluation strategy: if a fact skips the delta database (or delta concretization in case of $\mathrm{DL}_{\mathrm{DS}}$) and jumps directly to the total database, lack of all-τ rules means it could leave some of its consequences undiscovered by semi-naïve evaluation. Properties (1) and (2) of $\boldsymbol{\gamma}_\tau$ and $\boldsymbol{\gamma}_\Delta$ ensure that such issues are prevented.

The importance of property (2) also justifies us going through the semi-naïve semantics for $\mathrm{DL}_{\mathrm{DS}}$: it gives us insight into how custom data structures should behave in Byods, in particular, they must avoid the pitfall of having facts skip the delta phase.

4 FROM $\mathrm{DL}_{\mathrm{DS}}$ TO BYODS

In this section, we extend our core semantics ($\mathrm{DL}_{\mathrm{DS}}$) to account for several implementation-relevant concerns necessary to scale Byods to a mature implementation.

Datalog engines use several techniques for efficient evaluation of programs not covered by Datalog (or $\mathrm{DL}_{\mathrm{DS}}$ semantics); chief among them is indexing, which materializes multiple copies of a relation based on its usage in rule bodies. For example, given the rule `baz(x, y, z) :- foo(x, y), bar(y, z)`, a modern Datalog engine may materialize the indices $\{1\}$ for `foo` (i.e., an index on the second column), and $\{0\}$ for `bar` (i.e., an index on the first column). This allows the above rule to be operationalized via the following join plan:

```
for (y, xs) in foo_ind_1.iter_all() {
  if let Some(zs) = bar_ind_0.index_get(y) {
    for (x, z) in product(xs, zs) { baz.add(x, y, z); } } }
```

The example motivates the following design principle taken by Byods: Datalog engines interact with (read from and write to) a relation via its indices. In general, multiple logical indices will be necessary, and thus user-provided relation-backing data structures must expose multiple indices. Data structure choice may also depend on other factors, such as arity and column types of a relation (*e.g.*, an array-backed relation requiring columns to have integral types, an equivalence relation requiring columns to have the same type), or whether the data structure needs to support concurrent iteration and mutation for parallel evaluation of the Datalog program. For these reasons, rather than requiring relations to be tagged with specific data structures, Byods requires them to be tagged with what we call *data structure providers*. A data structure provider can take all the information relevant to a relation into account, and choose specific data structure(s) for the relation.

To enable the Byods compiler to communicate these implementation-relevant concerns to user-provided data structures—and thus enable optimal data structure selection— Byods defines a two-stage protocol. The first stage consists of a set of compile-time macros which the data structure provider must implement: these macros allow the data structure provider to perform compile-time customization based on static compile-time information. The second stage is concerned with the interaction of the Datalog engine and the data structures.

For the first stage, a data structure provider needs to implement a number of macros for various components required for a relation. For example, a macro named `rel_ind` decides on the type of a logical relation index, and another named `rel_ind_common` decides on the type of the data shared between all logical indices. The types returned by these macros correspond to the abstract domains (Ds) in $\mathrm{DL}_{\mathrm{DS}}$. With this protocol in place, our compiler calls these macros, providing all the information relevant to data structure selection mentioned above.

To give a concrete example, let's assume relation `foo` from the example above is defined like so: `#[ds(my_provider)] relation foo(Col0, Col1)`. This definition specifies that `foo` is backed by the data structure provider `my_provider`. A macro invocation in Byods for this relation looks like this:

```
my_provider::rel_ind_common!(
  foo,          // rel name
  (Col0, Col1), // column types
  [[1]],        // logical indices
  ser,          // parallel or serial
  (),           // user-specified params
)
```

Data structure providers can employ arbitrary logic when constructing a type for a relation or its indices. For example, a provider can provide structural sharing of logical indices of a relation [Subotić et al. 2018]. Index-sharing requires sophisticated logic, including graph algorithms. With our approach, we are in luck, we can use procedural macros in Rust to implement arbitrarily sophisticated logic for a provider. As the name implies, procedural macros are Rust functions invoked at compile-time in response to macro invocations.

As a final note on the first stage, data structure providers implemented using procedural macros sometimes find it useful to be able to inject code into the Datalog compilation, to, for example, define types implementing the traits required by Byods. They are given the opportunity to do so through the `codegen` macro that they must define. We use this capability in defining the `ind_share` (5.5) and `lat` (5.6) data structure providers.

For the second stage, our Datalog compiler expects the types constructed by data structure providers to implement a number of traits, through which it communicates with the data structures. Sharing data among logical indices is made possible by the indirection provided by the `ToRelIndex` trait. `rel_ind!`-returned types must implement this trait, whose functions `to_rel_index` and `to_rel_index_write` are supplied with the shared data, and whose return types must implement the relevant traits.

In Byods, the primary operations required of a logical index for querying its contents are key-based lookup (given a key, which is a tuple of the indexed-on columns, return an iterator over all the tuples corresponding to the key), and iteration over all the pairs of (key, value iterator)s. To support these operations, an index needs to implement the `RelIndexRead` and `RelIndexReadAll` traits respectively. These traits play a role analogous to the concretization function that we introduced in Section 3: the Datalog engine reads tuples represented by the abstract data structure for the relation via these traits. Naturally, all indices of a relation must agree on the same concretization for the Datalog program to behave sensibly. Another trait, `RelIndexWrite`, is responsible for injecting new facts into the data structure. This trait plays the role of the injection function from Section 3. Figure 1 provides the definitions for some of the key traits mentioned above.

Byods supports parallel evaluation of Datalog programs as well. When a Datalog program is to be evaluated in parallel, data structures need to also implement concurrent versions of the aforementioned traits. `CRelIndexRead` for example is the concurrent version of `RelIndexRead`. Parallelism in Byods relies on parallel iterator traits from the `rayon` crate. This both simplifies the task of implementing Byods traits for parallel data structures, as there exists a comprehensive library of combinators for parallel iterators that data structure providers can use, and ensures good utilization of multi core machines enabled by `rayon`'s work-stealing thread pools. Rather

```
pub trait RelIndexRead<'a> {
  type Key;  // indexed-on columns
  type Value; // remaining columns
  type IteratorType: Iterator<Item = Self::Value> + Clone + 'a;
  fn index_get(&'a self, key: &Self::Key) -> Option<Self::IteratorType>;
}
pub trait RelIndexReadAll<'a> {
  type Key: 'a; type Value;
  type ValueIteratorType: Iterator<Item = Self::Value> + 'a;
  type AllIteratorType: Iterator<Item = (Self::Key, Self::ValueIteratorType)> + 'a;
  fn iter_all(&'a self) -> Self::AllIteratorType;
  fn len_estimate(&'a self) -> usize;
}
pub trait RelIndexWrite {
  type Key; type Value;
  fn index_insert(&mut self, key: Self::Key, value: Self::Value);
}
pub trait RelIndexMerge {
  fn init(new: &mut Self, delta: &mut Self, total: &mut Self);
  fn merge(new: &mut Self, delta: &mut Self, total: &mut Self);
}
```

Fig. 1. Key traits through which Byods interacts with custom relations

than employing global locks when writing to indices in parallel, Byods requires data structures to implement `CRelIndexWrite`. The `index_insert` function of this trait takes a shared reference to `self` and therefore can be called concurrently by the Datalog engine. It is left to data structure providers to employ appropriate synchronization mechanisms to support parallel insertion. This strategy ensures data structures that can provide fine-grained locking (as employed by the default data structure provider in Byods) or lock-free insertion do not suffer from needless global locking.

In Byods relations must participate in semi-naïve evaluation, and as we explored in 3.1, a custom data structure must take care to populate its delta version correctly in particular. To that end, logical indices and the common index must implement the `RelIndexMerge` trait, whose `merge` function is provided with mutable references to all three versions of a relation index (`new`, `delta`, and `total`, where `new` is the version being written to in each iteration, and `delta` and `total` correspond to Δ and τ concretizations) and is responsible for correctly updating these versions (roughly speaking, that is joining `delta` into `total`, making sure `delta` has everything in `new`, plus everything that would be in total in the next iteration that isn't in `total` now, and finally clearing out `new`). From the point of view of the semi-naïve semantics of $\mathrm{DL_{DS}}$, `RelIndexMerge` prepares a relation (index) for Δ and τ concretizations.

The reader may have noticed that implementing a data structure provider is somewhat of an involved task. A user who wishes to implement a data structure for use in Byods must implement a number traits and then a number of macros for the data structure. One might prefer a simpler protocol, where implementing a trait or two would suffice for a data structure provider. This tension between flexibility and power on one hand, which our protocol is designed to provide, and simplicity and convenience on the other hand often exists in designing extensibility protocols for systems. Fortunately, this gap can be bridged. For example, we'll present a number of data structure providers specifically for binary relations in the next section. We've implemented an adaptor for binary relations: a type that implements the trait `ByodsBinRel` becomes a data structure provider with little more work. Using this approach saved us around 220 LOC when implementing the `trrel_uf` provider (5.4). We developed another abstraction, `BinRelToTernary`, that produces a ternary version of any binary relation (that implements `ByodsBinRel`) for free. We experiment with a ternary `trrel` in 5.3.

We demonstrate in the sections that follow the power and versatility of Byods. We define multiple data structure providers. They include `ind_share`, which provides index sharing for relations, but does not alter the semantics of a relation; `lat`, which extends Datalog with user-defined lattices, improving the expressivity of Datalog programs, including providing a framework for recursive aggregation; `trrel` and `eqrel`, which support transitive and equivalence relations respectively. The fact that Byods supports these data structure providers, some of which having little in common, shows the power of our approach.

5 EVALUATION AND APPLICATIONS

This section presents various data structure providers implemented in Byods, together with benchmark programs utilizing them. Our benchmarks show how Byods enables improving the performance of Datalog programs by tuning the data structures to the task at hand. We also present a data structure provider that extends the expressivity of Datalog, allowing us to write Datalog programs that would require recursive aggregation built-in to the Datalog engine otherwise.

Unless stated otherwise, the experiments presented in this section were run on a PC with an AMD Ryzen 9 4900H CPU and 32GB of RAM.

5.1 Equivalence Relations

The first benchmark uses equivalence relations. For this benchmark we implement specialized data structures for equivalence relations based on the union-find data structure [Galil and Italiano 1991; Galler and Fisher 1964]. High-performance equivalence relations in Datalog were first introduced in [Nappa et al. 2019]. These data structures can greatly improve the performance of Datalog programs requiring equivalence relations, including certain program analyses. Equivalence relations in Datalog are a good example of the benefits of the Byods approach. An equivalence relation, when implemented explicitly in Datalog, computes and stores all the tuples explicitly. For an equivalence class of size K, this means storing K^2 tuples. This is clearly suboptimal compared to specialized data structures for equivalence relations that avoid materializing the tuples and instead store the equivalence classes (that is, require $O(K)$ space per equivalence class).

In this benchmark, we have a relation `eq` that is seeded with a collection of facts of the form `eq(i, i + 1)` for $0 \leq i < N$ for some fixed N. The explicit version of the test program contains rules ensuring `eq` is an equivalence relation, while in the implicit version, `eq` is tagged with the `eqrel` backing data structure. This is a pathological case for the explicit version of the program, as it requires $\Omega(N^3)$ time to compute the equivalence closure of the input facts (there are $O(N^2)$ facts of the form `eq(a, b)`, and for each such fact, there are $O(N)$ facts of the form `eq(b, c)`). In contrast, the `eqrel` version uses a union-find-like backing data structure, and requires close to $O(N)$ time to compute the equivalence closure of the seeded facts. There is a significant improvement to space complexity as well: in the worst case, the explicit version of the program stores $O(N^2)$ tuples, while the `eqrel` version stores equivalence classes rather than individual tuples, and requires only $O(N)$ space.

Table 1 summarizes the running times and memory usage of these two versions of the program with different Ns. The results in the table bear out the $O(N)$ vs. $\Omega(N^3)$ difference in time complexity, and $O(N)$ vs. $O(N^2)$ difference in space complexity.

Table 1. Performance of `eqrel` vs. explicit (implemented via Datalog rules) equivalence relations

N	Time (s)		Memory (KiB)	
	`eqrel`	explicit	`eqrel`	explicit
100	0.000256	0.009	14	516
200	0.000397	0.077	28	2,032
400	0.001240	0.660	57	8,064
800	0.002420	14.670	114	32,129
1600	0.004700	191.000	228	128,259

```
// Analysis using explicit
// equivalence relations

relation vpt(Symbol, Symbol)
vpt(y, x), vpt(x, x) :- vpt(x, y).
vpt(x, z) :- vpt(x, y), vpt(y, z).
// ...
```

```
// Analysis using eqrel for
// equivalence relations
#[ds(eqrel)]
relation vpt(Symbol, Symbol)


// ...
```

Fig. 2. Steensgaard analysis via explicit rules (left) and our eqrel relations (right).

5.2 Steensgaard Analysis

Steensgaard analysis [Steensgaard 1996] is a near-linear time pointer analysis when implemented using union-find data structures. The analysis efficiently merges flow-sets of variables at assignment points. This merging deliberately sacrifices precision to improve performance. The same tradeoff would not be possible in Datalog without a specialized data structure for equivalence relations, since, as we discussed, computing equivalence relations in Datalog explicitly is very costly.

Soufflé's authors benchmarked their implementation of equivalence relations via a Steensgaard analysis of OpenJDK [Nappa et al. 2019]. We implemented an equivalent analysis in Byods and replicated the same experiment. An exemplary rule is shown in Figure 2: the explicit version materializes an equivalence in the the `vpt` relation (standing for "variable points to"); the optimized version is simply tagged with `eqrel`.

Because our results are due to the same issues studied in section 5.1, we elide a detailed comparison. We did, however, replicate the evaluation to compare our eqrel performance against Soufflé on one thread. As we expected, we saw very significant speedups compared to the explicit version of the analysis, which would take over ten hours either in Soufflé or Byods. Furthermore, we saw similar results to Soufflé's eqrel, with a median run time (five runs) of 120ms in Byods vs. 170ms in Soufflé. We also verified that both analyses produced identical outputs.

5.3 Transitive Relations

Computing transitive closures is a common task in Datalog, and required by various program analyses. A relation `r` can be made transitive in Datalog by inclusion of the rule `r(x, z) :- r(x, y), r(y, z)`. This incurs a time complexity of $O(N^3)$ in the worst case, where N is the number of tuples in the relation before it is made transitive. There is a transformation that improves the time complexity of making a relation transitive to $O(N^2)$: it requires a whole-program rewrite that replaces appearances of `r` in rule heads with a new relation `r_proto`, and replaces the

Table 2. Rust borrow checker experiments for the transitive relation and index sharing data structure providers. `ind_share` is introduced in 5.5. Benchmark programs are from [Sahebolamri et al. 2022].

Program	LOC	Time (s)					Memory (MiB)		
		explicit	`trrel`	speedup	`ind_share`	speedup	explicit	`trrel`	`ind_share`
clap-rs	2100	8.5	5.1	1.7x	9.65	0.88x	621	328	414
serde-fmt	170	3.74	1.77	2.1x	4.00	0.93x	483	311	360
ascent-codegen	800	1.38	0.65	2.1x	1.14	1.21x	182	125	148
polonius_comp	1000	32	6.2	5.2x	15.8	2.02x	1461	768	1034
chess-search	600	39	14.5	2.7x	26	1.50x	2879	2224	2344

above rule with `r(x, z) :- r_proto(x, y), r(y, z)`. Manually performing this task could be somewhat laborious and error-prone. What's more, the addition of another relation increases the space requirements by a constant factor.

We take advantage of our approach and implement `trrel`, a provider for transitive relations. `trrel` improves the time complexity of making a relation transitive compared to the naïve way of doing so, and helps eliminate the manual transformation described above. Note that unlike `eqrel`, `trrel` is backed by normal data structures for relations, and provides asymptotic improvements in time-complexity only compared to the naïve approach to making a relation transitive. `trrel` however can still provide constant-factor improvements compared to the whole-program-rewrite approach to making a relation transitive.

To test `trrel`, we use Polonius, a Datalog-based implementation of the Rust borrow checker [Matsakis and RustDevelopers 2023]. Borrow checking is a static analysis done by the Rust compiler to ensure references (*borrows*) in Rust code are valid when dereferenced, a crucial feature of the Rust language that contributes to its memory-safety [Weiss et al. 2019]. Polonius contains a ternary relation `subset(point, origin1, origin2)` whose selection on every `point` is made transitive by inclusion of the following rule:

```
subset(p, o1, o3) :- subset(p, o1, o2), subset(p, o2, o3).
```

This is the most taxing relation in Polonius, and optimizing its storage and computation can lead to potentially significant speedups.

Our `trrel` implementation is capable of handling ternary relations like `subset`. We compare an implementation of Polonius with `subset` backed by `trrel` vs. a version with the above explicit rule for `subset` in Table 2. As the results demonstrate, using `trrel` noticeably improves the performance of the analysis in most cases, with speedups of up to 5×. This improvement is partially due to algorithmic improvements in computing transitive closures, and it is partially due to structural sharing of different indices of the `subset` relation made possible by our approach. We also observe lower memory consumption as a result of structural sharing of indices.

5.4 Union-find Based Transitive Relations

Following the Byods philosophy that one size does not necessarily fit all, besides `trrel`, we implemented an alternative data structure provider for transitive relations: `trrel_uf`. This provider is based on ideas from the union-find data structure. To motivate `trrel_uf`, we note that a binary relation, when thought of as a directed graph, can be represented as a DAG (directed acyclic graph) of its SCCs (strongly connected components). In case the graph has a relatively small number of relatively large SCCs, its transitive closure is best stored as (the transitive closure of) the DAG of

Table 3. Transitive closure experiments on real-world graphs. `trrel_uf` is backed by `TrRelUnionFind`, our union-find based data structure for transitive relations. Graphs are from [Leskovec and Krevl 2014]. OOM indicates that the program ran out of memory.

Graph			**Time (s)**				**Memory (MiB)**			
Name	Edges	TC size	`trrel_uf`	`trrel`	explicit	Soufflé	`trrel_uf`	`trrel`	explicit	Soufflé
email-Eu	26K	793K	0.022	0.509	0.844	1.5	1.93	13.4	31.7	39.3
Wiki-Vote	104K	12M	1.8	11.9	10.2	15.0	145	182	533	469
HepTh	52K	74.6M	6.0	51	44	80	49	1222	3590	2408
ca-AstroPh	396K	320M	39	600	792	595	174	5074	15361	12328
BrightKite	428K	3.2B	775	OOM	OOM	OOM	2382	OOM	OOM	OOM

equivalence classes, rather than an explicit list of all connected nodes, where each equivalence class corresponds to an SCC of the graph.

Based on this idea, we implemented the `TrRelUnionFind` data structure, which backs `trrel_uf`. `TrRelUnionFind` only stores connections between equivalence classes (SCCs) explicitly, and when two (or more) equivalence classes are to be unified (i.e., when a new fact (x, y) is to be added, where there is already a connection from y's equivalence class to x's), it uses the union-find technique of marking one class as the parent of the other to unify the classes, and updates the stored connections between classes, ensuring they always point to up-to-date class ids.

This technique can provide significant improvements in time and space complexity, but only if the structure of the underlying graph is conducive to this technique. In one extreme, if a graph with N nodes is a single SCC, our data structure can require only $O(N)$ space and time to compute its transitive closure, a substantial improvement over the usual Datalog-based solution (or `trrel`), which requires $O(N^2)$ space and time. In the other extreme, if the graph has N SCCs (i.e., it is a DAG with N nodes), `TrRelUnionFind` provides no asymptotic improvements, it will revert to $O(N^2)$ space and time (in the worst case), with some constant overhead compared to the Datalog-based solution.

Based on this discussion, `trrel_uf` can be a good option for computing the transitive closure of graphs of social networks, which usually contain communities of non-trivial sizes. To evaluate `trrel_uf`, we compute the transitive closures of a number of real-world graphs, and compare the `trrel_uf` results with the results of explicit rules for computing the transitive closure of a relation; we include `trrel` and an implementation of transitive closure in the Soufflé Datalog engine [Jordan et al. 2016] as well for comparison. The results, presented in Table 3 demonstrate significant gains in time and memory when using `trrel_uf`. We observe a consistent speedup of around an order of magnitude, and slightly to very significantly ($\sim 30\times$) lower memory usage.

5.5 Index Sharing

As we discussed, efficient evaluation of Datalog requires indexing relations as demanded by their use in rules. Typically a relation requires multiple indices, as it is joined in different ways in rules. Storing one physical index per required logical index can be both space- and time-inefficient. Recognizing that one physical index can serve multiple logical indices, we can reduce the space required for Datalog computations. As updating and combining an index is time-consuming, this can also help speed up Datalog computations in some instances.

The key to index sharing is to allow a physical index to have a curried form. For example consider a scenario where a relation $r(A, B, C)$ requires two logical indices, one on columns A, B, and the other on column A. In this scenario a single physical index of the form $A \to B \to^* C$ could serve both these logical indices. Here, $\rightarrow$ indicates a map data structure, such as a hash map, and $\rightarrow^*$ indicates a multi-valued version of this data structure. [Subotić et al. 2018] presents an algorithm for picking the minimum number of physical indices required to cover a set of logical indices.

We implemented `ind_share`, an index-sharing data structure provider for Byods based on this idea. We employ the algorithm presented in [Subotić et al. 2018] for automatic selection of the minimum number of physical indices required to cover a set of logical indices. Our implementation also allows the user to explicitly ask for specific physical indices. This is useful when there is more than one minimal set of physical indices covering the required logical indices, and one choice of physical indices is superior (e.g., heavily used logical indices provide better performance if backed by simpler physical indices).

In our implementation of index sharing, we use hash maps as the underlying data structure (unlike [Subotić et al. 2018], which uses B-trees). Using hash maps complicates the implementation because unlike B-trees, which naturally allow prefix lookups through range queries, using hash maps requires building up nested hash map structures (for example `HashMap<A, HashMap<B, Vec<C>>>` for the abstract physical index $A \rightarrow B \rightarrow^* C$, from the example above). The main advantage of taking this approach as opposed to using B-trees is that stored keys are not required to be total orders. A big selling point of embedding a Datalog variant in Rust is that the user can utilize arbitrary Rust data types in their Datalog programs (including data types representing ASTs, binding environments, evaluation contexts, states of abstract machines for program analysis, etc.), and limiting these types by requiring them to be total orders would undermine this selling point. Nevertheless, Byods permits implementing a B-tree-backed index-sharing data structure provider as well.

To support the index-sharing data structure provider, we developed a library for manipulating and adapting physical indices to the required logical index forms. This library revolves around the traits `DictRead` and `MultiDictRead`, representing single-valued and multi-valued maps respectively. We developed adaptor types for transforming the shape of a physical index. For example, the type `DictOfDictUncurried` transforms a map of the form $A \rightarrow B \rightarrow C$ (i.e., a type implementing `DictRead<Key = A, Value: DictRead<Key = B, Value = C>>`) to a map of the form $(A, B) \rightarrow C$ (i.e., `DictRead<Key = (A, B), Value = C>`).

To evaluate the index-sharing data structure provider, we again use the Polonius benchmarks. Table 2 also includes a version of the Rust borrow checker where the most taxing relation in the Datalog program, `subset`, is tagged with `ind_share`. This relation requires 5 logical indices; using index-sharing, these 5 logical indices can backed by 2 physical indices, highlighting the benefits of index-sharing. As the results demonstrate, using the index sharing strategy helps improve the performance of the analysis in many cases. We also observe improvements in memory use. The memory use of the `ind_share` version is consistently lower than the default version.

Comparing the index-sharing strategy results with the results of using the `trrel` specialized data structures also highlights the advantages of each one. `trrel` is a specialized data structure for transitive relations, providing best performance for this application. The index-sharing data structure provider on the other hand is a general-purpose data structure provider that can be applied to any relation in the Datalog program.

We finally note that the fact that we could implement the index-sharing data structure provider in Byods is a testament to its generality and its capabilities. Implementing this provider requires graph algorithms (e.g., Gabow's algorithm for the maximum matching problem in graphs [Gabow 1976]), in addition to employing logic for synthesis of `DictRead` and `MultiDictRead` transformer types to adapt a physical index to its required logical forms, which are made possible by our framework and Rust's procedural macros.

5.6 Lattices in Datalog

Datalog evaluation can be viewed as ascending the powerset lattice to a fixed point (of the immediate consequence operator for the program, as discussed in Section 3). It should come as no surprise that some applications require computing fixed points of different kinds of lattices. Flix [Madsen et al. 2016] pioneered a particular approach to solving this problem: defining lattices in addition to relations in a Datalog program. In Flix, a table `l(K, V)` defined as a `lat` rather than a relation is semantically a map (join-semi-)lattice as opposed to a set of tuples. A map lattice is a partial map $K \rightharpoonup V$ whose least upper bound operation is defined point-wise: $(m_1 \sqcup m_2)(x) = m_1(x) \sqcup m_2(x)$. By convention, V is the last column of the table and K is the tuple type corresponding to the other columns. For lattices to be an effective addition to Datalog, one needs the ability to define new lattice data types for V. In Flix, this is achieved by inclusion of a pure functional language; as we'll see, in Byods the user can define lattice data types in Rust.

This style of lattices extends the expressivity of Datalog, expanding the domain of applicability of Datalog programs. For example, lattices allow more useful program analyses to be expressed in Datalog, including constant-propagation analyses, strong update analyses, and the like [Madsen et al. 2016; Sahebolamri et al. 2022].

To show the versatility of our approach, we use Byods to define a `lat` data structure provider. `lat` works similarly to lattices in Flix. A relation tagged with `lat` is a partial map that is strongly updated whenever a new fact is to be added to it whose key is already present in the map. As we see in the evaluation that follows, having a parallelized `lat` could result in remarkable speedups in certain applications of Datalog plus lattices.

A notable (and less explored) example of use of lattices in Datalog is programs that require recursive aggregation. Recursive aggregation (as opposed to stratified aggregation, which is supported in many Datalog engines) requires updating the aggregated relation as part of evaluating the aggregation result. Allowing Datalog to express recursive aggregation has been an active area of research [Mazuran et al. 2013; Ross and Sagiv 1992; Zaniolo et al. 2017]. Lattices in Datalog both support recursive aggregation, and help eliminate the air of mystery around monotonic (recursive) aggregates, helping Datalog users better understand programs requiring recursive aggregation.

One example of recursive aggregation is counting the number of distinct paths between pairs of nodes in a graph. This program can be expressed in DeALS, a Datalog engine that supports recursive aggregation [Shkapsky et al. 2015], as follows:

```
cpaths(X, Y, mcount<X>) <- edge(X, Y).
cpaths(X, Y, mcount<(Z, C)>) <- cpaths(X, Z, C), edge(Z, Y).
countpaths(X, Y, max<C>) <- cpaths(X, Y, C).
```

The monotonic aggregate `mcount` in this example can be thought of as being implemented using a map lattice (we are referring to the inner lattice, `cpaths` itself would be a map lattice of a map lattice!). The atom `cpaths(X, Z, mcount<(Y, C)>)` in a rule head joins (as in "takes the least upper bound of") the existing map lattice for the relation `cpaths` at point `(X, Z)` with the map lattice defined on a single point: `{Y -> C}`. When `cpaths` appears in a rule body (as in `cpaths(X, Y, C)`), the meaning of the aggregated column changes; now, the variable bound to the aggregated column (`C`) refers to the sum of the values stored in the map lattice. The same program can be expressed in Datalog with lattices:

```
relation edge(Node, Node)
#[ds(lat)] relation cpaths(Node, Node, MapLattice<Node, usize>)
relation countpaths(Node, Node, usize)

cpaths(x, y, map_lat![(x, 1)]) :- edge(x, y).
```

Table 4. Graphs used to test PageRank, from [Leskovec and Krevl 2014].

Graph			Time (s)
Name	Nodes	Edges	1 thread
Slashdot	77K	905K	26.3
BerkStan	685K	7.6M	77.2
Orkut	3.1M	117M	838
Pokec	1.6M	30M	904
LiveJournal	4.3M	69M	2004

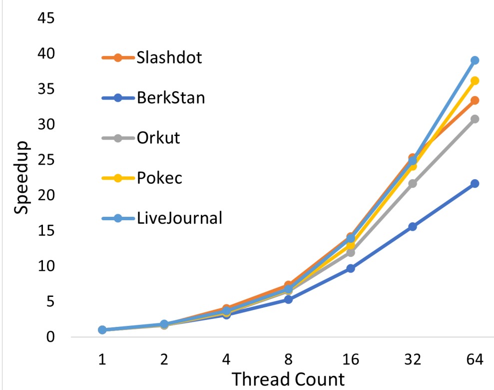

Fig. 3. Scaling PageRank: 1 – 64 threads

```
cpaths(x, y, map_lat![(z, c.sum())]) :- cpaths(x, z, c), edge(z, y).
countpaths(x, y, c.sum()) :- cpaths(x, y, c).
```

In this example, `MapLattice` is a user-defined Rust data type (again, not to be confused with the map lattice semantics of the `lat` data structure provider). This example demonstrates how straightforward it is to translate programs requiring recursive aggregation to Datalog programs with lattices. Having translated the example, we may find it easier to interpret what the program does. It simply states that the number of paths from `x` to `y` is the sum of the number of paths from `x` to `z`, where `z` is directly connected to `y` (plus one if there is a direct edge from `x` to `y`).

Another typical example of the power of recursive aggregation is the PageRank algorithm [Page et al. 1999]. We can implement PageRank in Byods straightforwardly as follows (implementation adopted from [Wu et al. 2022]):

```
relation matrix(Node, Node, Num)
#[ds(lat)] relation rank(Node, MapLattice<Node, Num>)

rank(x, map_lat![(x, i)]) :-
  matrix(x, _, _), let i = (1.0 - ALPHA) / vnum.
rank(x, map_lat![(y, k)]) :-
  rank(y, c), matrix(y, x, w), let k = ALPHA * c.sum() * w.
```

Here `matrix` is the graph's adjacency matrix. `matrix(x, y, w)` indicates there is an edge from `x` to `y` with weight (probability) `w`. `vnum` is the number of vertices in the graph. `ALPHA`, a damping factor, is a constant that affects the convergence rate of the query; it is generally (and in our evaluation) set to 0.85. Last, the `sum` function on `MapLattice` returns the sum of the stored values.

We used the above program to benchmark the performance of `lat` in Byods. Our implementation supports Byods's parallel contracts, enabling our program to take advantage of multi-core CPUs. For this evaluation, we use a number of large real-world graphs from [Leskovec and Krevl 2014] (shown in Table 4). We ran our experiments on a workstation with an AMD EPYC 7713P 64-Core CPU and 512 GB of RAM. Single-thread timings, taken as a baseline, are also shown in Table 4. Figure 3 details speedup vs. thread count for each experiment. Our experiments show that our implementation scales well, achieving both strong scaling (across thread counts) but also weak scaling: scalability roughly increases with problem (graph) size.

5.7 Analysis of LLVM

To demonstrate the usability and maturity of Byods, we implemented a realistic whole-program pointer analysis for LLVM IR. Our analysis uses the standard allocation-site abstraction, reducing the potentially-unbounded set of run-time allocations to the static, finite set of allocation-site labels. The overall structure is inclusion-based (Andersen-style [Andersen 1994]), and supports k-callsite context-sensitivity for arbitrary k. Our prototype analysis is array-, field-, and flow-insensitive, and doesn't yet support heap cloning. Our implementation of this analysis consists of roughly 800 lines of rules in Byods, along with roughly 3400 lines of supporting code (mainly parsing LLVM modules).

Correctness, performance, and ease of implementation are all essential for such an analysis. Byods allows us to implement our analysis by writing rules that closely mirror traditional formalizations of Andersen-style analyses [Bravenboer and Smaragdakis 2009]. At the same time, our implementation also inherents all of the parallelism of Byods and utilizes index sharing to mitigate the significant time and memory costs associated with precise, context-sensitive static analysis. On a more pragmatic level, Byods provides an ideal setting for rapidly prototyping such analyses due being an EDSL embedded in Rust. k-limited contexts are represented using standard Rust data structures (*e.g.*, using a double-ended deque, rather than manual monomorphization of rules), obviating the need to construct a plethora of analysis variants for each supported value of k as is done in analyses written in other Datalog variants, including Doop [Bravenboer and Smaragdakis 2009]. Programs can be ingested into the analysis using off-the-shelf libraries rather than custom, standalone programs that generate Datalog facts [van Tonder 2021]; our analysis uses the popular `llvm-ir` Rust crate (library). Finally, Byods allows for experimenting with novel data structures to achieve satisfactory performance/precision trade-offs. We implemented a variant of our analysis based on `trrel_uf`, though it does not result in performance gains for our current implementation, as our workloads generated large numbers of small equivalence classes. We plan to improve precision by reasoning about out-of-bounds array accesses using a constant propagation analysis that utilizes Byods's support for lattices.

LLVM IR is a ubiquitous intermediate representation (IR) used in dozens of compilers, including Clang, GHC, and rustc [Lattner and Adve 2004]. LLVM programs consist of functions, which are made up of basic blocks, which are in turn composed of instructions in SSA form. Instructions store their result (if any) to a virtual register. Instructions have operands, which may be constants, formal parameters of the surrounding function, or virtual registers. Intra-procedural control-flow between basic blocks is structured using primitive operations such as conditional branches. Inter-procedural control-flow is mediated by a small number of instructions such as `call`. These instructions accept a function pointer, allowing for indirect (computed) calls.

While LLVM has an extensive set of instructions, only a few are relevant to a pointer analysis. The `alloca` instruction creates a new allocation on the stack. The `phi` instruction implements ϕ-nodes in the SSA form. The `call` instruction transfers control to another function, mapping actual parameters to formal parameters; `return` returns a value to the caller.[7] `load` reads from memory, and `store` writes to it. `getelementptr` is used to add an offset to a base pointer. It is used to e.g., load a specific index from an array or access a particular field of a struct. Our analysis is neither array- nor field-sensitive, so it treats `getelementptr` as a no-op. The analysis does not yet handle instructions pertaining to vectorized execution nor exception handling. The set of allocation-site labels consist of global variables, `alloca` instructions, direct calls to a set of predetermined functions such as `malloc`, and a special `Top` allocation, described below.

[7] For the sake of simplicity, we'll ignore the other `call`-like instructions, they are handled almost identically by the analysis.

Following cclyzer [Balatsouras and Smaragdakis 2016], the analysis treats allocations differently from virtual registers, which cannot be addressed by pointers. The output of the analysis consists of the following relations:

- `operand_points_to` relates operands to the (abstract) allocations they may point to.
- `alloc_points_to` relates allocations that may contain pointers to the allocations the pointers may point to.
- `calls` relates call-like instructions to the functions they may call.

Unlike cclyzer, the analysis does not attempt to recover a notion of type for each allocation. The LLVM community has realized that the intended memory model is inherently untyped; compiler optimizations based on notional allocation types have been found to be unsound. The language will soon drop support for typed pointers [LLVM-Authors 2023].

To make a respectable attempt towards soundness, a pointer analysis of LLVM must handle dozens of language-specific details. Our analysis supports many such trivialities, such as explicitly modeling `memcpy`; proper initialization of the `argv` and `envp` arguments to a C program's `main` function; modeling special, pre-initialized global variables like `stdin`, `stdout`, and `stderr`; and support for a variety of allocation functions including not only `malloc`, but also `realloc`, `calloc`, explicit calls to libc's `alloca` function (as contrasted with LLVM `alloca` instructions), and `_Znwm` (the mangled name of C++'s `new` operator).

More substantially, the analysis supports user-provided *function signatures*, which provide sound approximations of externally-defined functions. Calls to externally-defined functions may arise from use of a dynamically-linked library, or libc. For example, the signature `{ "return-aliases-arg": { "arg": 0 } }` is used to model `memchr`; it states that the return value is a pointer that may alias the function's first argument. The analysis includes 150 such signatures for common external functions. If the analysis encounters a call to an external function without a corresponding signature, it treats the return value as a pointer to a special `Top` allocation. This allocation signals that a pointer could point to *any* other allocation in the program. Loading from `Top` yields `Top`; storing to `Top` is a no-op.

We evaluted our analysis on a number of real-world programs, presented in the leftmost column of Table 5. We evaluated using our AMD EPYC workstation (64 core, 512GB) from 5.6. We ran our analysis with either one or two choices for k (larger k makes the state space polynomially-larger). We used six programs: Jackson is a small IRC client, Lua and Luac are the interpreter and compiler for Lua, httpd is an HTTP server, SQLite is a SQL database engine, and Redis is an in-memory database. We ran two versions of our analysis, one using the `ind_share` data structure provider for the largest relations in the program, and the other using the default data structure provider. We chose two settings for k in the first four programs, but analyze SQLite and Redis using only 0-callsite sensitivity; 1-callsite sensitivity for these programs resulted in OOMs.

Our results reveal several broad trends. First, both analyses scale relatively well, though diminshing returns are apparent at 64 threads; however, we believe it is promising that scalability increases as state space increases. Second, the `ind_share` version consistently uses significantly less memory compared to the default version; indeed, 1-callsite sensitivity on httpd does not even terminate without index sharing. While the default version is often faster in low thread counts, the `ind_share` version appears to exhibit better scaling, and sometimes overtakes the default version at higher thread counts. We believe this is because the `ind_share` version writes to fewer indices, inducing less lock contention. Once again, the fact that we could improve scaling simply by swapping to a different data structure provider speaks to the robustness of the Byods approach.

Table 5. LLVM IR pointer analysis experiments. k is the context sensitivity of the analysis. Pts is the size of `operand_points_to`, the biggest relation computed by the analysis. ind is the version of the analysis using `ind_share`, and def is the version using the default data structure provider.

Prog.	k	Pts	Time (s) by thread count								Memory (MiB)	
			8		16		32		64			
			ind	def	ind	def	ind	def	ind	def	ind	def
Jackson	3	10.2M	8.8	10.9	7.6	10.8	7.4	10.1	7.3	9.7	1,940	3,107
	4	164M	128	166	102	164	99	152	94	143	30,466	50,533
Luac	0	3.2M	5.2	4.9	3.8	4.4	3.2	4.4	3.3	5.0	555	732
	1	45M	68	58	48	52	45	50	52	50	3,798	8,000
Lua	0	12.8M	19.6	16.5	13.0	14.6	10.4	14.6	9.7	15.8	1,782	3,222
	1	214M	303	257	193	215	154	199	183	201	14,074	34,335
httpd	0	27.1M	154	103	91	72	65	59	58	61	3,545	6,100
	1	5.39B	26,530	OOM	15,240	–	10,285	–	9,793	–	425,000	OOM
SQLite	0	167M	830	540	471	367	312	284	248	326	26,665	44,597
Redis	0	735M	19,083	11,536	9,210	6,486	5,424	4,087	4,063	3,541	99,500	178,264

6 RELATED WORK

There are several threads of related work.

Tuple Representations for Datalog. Datalog has seen repeated resurgences in interest, often coinciding with novel developments in data structure representation. This excitement is often due Datalog's application to program analysis and related fields. For example, bddbddb demonstrated the scalability gains to be had from BDD-based tuple representations. Such BDD-based representations have fallen out of favor due to representation-imposed blowups and the need for variable orderings. LogicBlox provided the next innovation, this time leveraging an optimized join strategy, Leapfrog Triejoin [Aref et al. 2015; Veldhuizen 2014]. LogicBlox enabled the DOOP pointer analysis [Bravenboer and Smaragdakis 2009], which was subsequently ported to the Soufflé solver, delivering impressive speedups [Antoniadis et al. 2017]. Soufflé contains several tuple representations, including concurrent B-trees and a novel "Brie" data structure. Brie's purpose is to represent high-density relations, and leverages principles from both B-trees and tries to achieve this [Jordan et al. 2019]. Like tries, a Brie performs prefix-deduplication (for tuples, in case of Datalog), and like B-trees, a Brie uses cache-friendly arrays of multiple values per node. More recently, Soufflé has unified its data structures via the *Datalog-Enabled Relational* (DER) approach [Jordan et al. 2022]. DERs must support insertion, iteration, range lookup, and membership and emptiness checking. Byods may be seen as systematizing and generalizing the DER approach, providing a semantic (rather than merely mechanical) account of how to integrate user-provided data structures with Datalog's compilation and semi-naïve semantics. Byods enables sophisticated, semantics-altering data structure providers that have a holistic view of the relation they are supporting, in contrast to the DER approach, which enables alternative implementations of a B-tree-like interface. Byods is additionally distinguished from DER by our macro-based approach, enabling the user to build data structures directly in Rust without the need to manually extend Soufflé's implementation (a nontrivial effort).

Programming with Union-Find Data Structures. Union-find data structures form crucial components in applications such as type inference and, more recently, equality saturation [Tate et al. 2009; Willsey et al. 2021]. Motivated by its inability to realize algorithms requiring union-find, Soufflé has recently added eqrel relations, which provide union-find to users of Souffé [Nappa et al. 2019]. While Soufflé does generate code that is compatible with a variety of engine-provided data structures, it provides no points for user extension; Byods builds upon a formal extension of Datalog, $\mathrm{DL}_{\mathrm{DS}}$, to enable the user to bring their own data structures. Last, egglog has recently provided a new platform which allows users to mix Datalog and equality saturation [Zhang et al. 2023]; we plan to study how Byods may be used to implement equality saturation in future work.

Programming with Fixed Points over Non-Powerset Lattices. Datalog's restriction to the sets-of-facts interpretation imposes severe limitations, forbidding the expression of common idioms in program analysis (*e.g.*, the constant propagation or interval lattice), graph analytics, and similar applications. Such restrictions represent expression-limiting pain points for engineers of such analyses in Datalog, and successful applications of Datalog have harmoniously leveraged the powerset lattice, eliding more general non-powerset lattices. One popular line of work is that of *recursive aggregation*, which puts Datalog rules in a loop with a lattice join operator (which, in general, operates over non-powerset lattices) [Ross and Sagiv 1992]. This generalization of Datalog from powersets to arbitrary lattices remediates the overhead from powerset-based encodings.

Datalog^{FS} [Mazuran et al. 2013] and subsequently DeALS [Shkapsky et al. 2015] both target high-throughput graph analytics applications which involve recursive aggregation (e.g., single-source shortest paths). This line of work continued with several other engines evolving suit the needs of large-scale social-media and graph analytics, including BigDatalog [Shkapsky et al. 2016], RaSQL [Wang et al. 2020], and DcDatalog [Wu et al. 2022]. These systems are primarily aimed at processing extremely large graphs on a cluster of nodes, either using commodity-grade network infrastructure (e.g., Apache Spark) or novel parallel approaches targeted at large unified nodes. We elide a detailed comparison with the distrubted tools, though initial experimentation shows favorable scaling (vs. DeALS and RaSQL) against these systems on a single node. DcDatalog supports recursive aggregation and targets large unified nodes, but is closed-source; reported data from DcDatalog's paper suggests it scales roughly similarly to Byods for PageRank.

Several authors have explored the semantic ramifications of Datalog's extension to lattices. Flix extends Datalog to a rich combination of functional and logic programming, supporting (non-powerset) lattices [Madsen et al. 2016]; we elide a detailed comparison against Flix, limited experiments suggest that Byods' efficient compilation and parallel implementation far outperformed equivalent Flix implementations. Similarly, Datafun generalizes Datalog to a theory of programming with monotonic maps over join-semilattices [Arntzenius and Krishnaswami 2019, 2016]. That work formalizes a category-theoretic denotational semantics, and provides a rich type system, ensuring the termination of well-typed programs. Our contributions are largely orthogonal, focused on harmonizing Datalog's operationalization as joins with user-provided data structures.

Datalog as an Embedded DSL. The EDSL-based strategy for implementing Datalog is becoming popular, offering many attractive benefits, chiefly the interoperability with constructs from the host language. Byods is a Rust EDSL, as are Ascent [Sahebolamri et al. 2022] and Crepe [Zhang 2023]; Racket also includes a Datalog [McCarthy 2022]. Our focus is on harmonizing user-provided data structures with efficient compilation using state-of-the-art methods, and thus we see our work as orthogonal to the current state of the art, and a natural extension to this line of work.

7 CONCLUSION

Modern Datalog engines are extremely enticing tools for the implementation of deductive-analytic workloads due to the price-point they deliver, marrying declarative specification with high-performance compilation. Unfortunately, would-be Datalog users confront a key challenge: if they need to represent tuples using a special-purpose data structure (*e.g.*, union-find), their only option is to extend a state-of-the-art engine with their data structure (an imposing challenge).

We propose the "Bring Your Own Data Structures" approach, wherein declarative rules are compiled to join plans which operate harmoniously with user-provided data structures. This is accomplished by emitting code which interacts with relations by means of a concretization function, which interfaces between the compiled rules and user-provided data structures. We formalize our ideas in $\mathrm{DL}_{\mathrm{DS}}$, an extension of Datalog to accomodate user-provided data structures, and articulate the formal properties demanded by the Byods approach. This formalism is the basis for a mature implementation, Byods, which provides a macro-based protocol to interface user-provided data structures with join plans compiled from rules. We implement our approach as a macro-embedded DSL in Rust. We conduct a large variety of evaluations which measure our implementation's robustness and scalability. Exciting results include speed and memory gains for realistic applications (the Polonius implementation of Rust's borrow checker), strong and weak scaling for lattice-oriented programming (PageRank) and large-scale program analysis (LLVM), and crucial algorithmic optimizations necessary to implement state-of-the-art deductive-analytic workloads.

8 ACKNOWLEDGEMENTS

This work was funded in part by NSF PPoSS planning award CCF-2217037. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. N66001-21-C-4023. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA.

REFERENCES

Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph. D. Dissertation. DIKU, University of Copenhagen.

Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. 2017. Porting doop to soufflé: a tale of inter-engine portability for datalog-based analyses. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. ACM, 25–30.

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 *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1371–1382. https://doi.org/10.1145/2723372.2742796

Michael Arntzenius and Neel Krishnaswami. 2019. Seminaïve Evaluation for a Higher-Order Functional Language. *Proc. ACM Program. Lang.* 4, POPL, Article 22 (dec 2019), 28 pages. https://doi.org/10.1145/3371090

Michael Arntzenius and Neelakantan R Krishnaswami. 2016. Datafun: a functional Datalog. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 214–227. https://doi.org/10.1145/2951913.2951948

George Balatsouras and Yannis Smaragdakis. 2016. Structure-sensitive points-to analysis for C and C++. In *Static Analysis: 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings 23*. Springer, 84–104.

François Bancilhon. 1986. *Naive Evaluation of Recursively Defined Relations.* Springer New York, New York, NY, 165–178. https://doi.org/10.1007/978-1-4612-4980-1_17

Francois Bancilhon and Raghu Ramakrishnan. 1986. An Amateur's Introduction to Recursive Query Processing Strategies. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data* (Washington, D.C., USA) *(SIGMOD '86)*. Association for Computing Machinery, New York, NY, USA, 16–52. https://doi.org/10.1145/16894.16859

Aaron Bembenek, Michael Greenberg, and Stephen Chong. 2020. Formulog: Datalog for SMT-based static analysis. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–31. https://doi.org/10.1145/3428209

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. 243–262.

Stefano Ceri, Georg Gottlob, Letizia Tanca, et al. 1989. What you always wanted to know about Datalog(and never dared to ask). *IEEE transactions on knowledge and data engineering* 1, 1 (1989), 146–166. https://doi.org/10.1109/69.43410

Harold N. Gabow. 1976. An Efficient Implementation of Edmonds' Algorithm for Maximum Matching on Graphs. *J. ACM* 23, 2 (apr 1976), 221–234. https://doi.org/10.1145/321941.321942

Zvi Galil and Giuseppe F. Italiano. 1991. Data Structures and Algorithms for Disjoint Set Union Problems. *ACM Comput. Surv.* 23, 3 (sep 1991), 319–344. https://doi.org/10.1145/116873.116878

Bernard A. Galler and Michael J. Fisher. 1964. An Improved Equivalence Algorithm. *Commun. ACM* 7, 5 (may 1964), 301–303. https://doi.org/10.1145/364099.364331

Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 422–430. https://doi.org/10.1007/978-3-319-41540-6_23

Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. 2019. Brie: A Specialized Trie for Concurrent Datalog *(PMAM'19)*. Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/3303084.3309490

Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. 2022. Specializing parallel data structures for Datalog. *Concurrency and Computation: Practice and Experience* 34, 2 (2022), e5643. https://doi.org/10.1002/cpe.5643

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.

Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

LLVM-Authors. 2023. *Opaque Pointers – LLVM documentation.* Retrieved April 11, 2023 from https://llvm.org/docs/OpaquePointers.html

Nuno Lopes, Nikolaj Bjørner, Nick McKeown, Andrey Rybalchenko, Dan Talayco, and George Varghese. 2016. Automatically verifying reachability and well-formedness in P4 Networks. *Technical Report, Tech. Rep* (2016).

Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From datalog to flix: A declarative language for fixed points on lattices. *ACM SIGPLAN Notices* 51, 6 (2016), 194–208. https://doi.org/10.1145/2908080.2908096

Nicholas Matsakis and RustDevelopers. 2023. *Rust-Lang/polonius: Defines the Rust borrow checker.* Retrieved April 14, 2023 from https://github.com/rust-lang/polonius

Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. *Ada Lett.* 34, 3 (oct 2014), 103–104. https://doi.org/10.1145/2692956.2663188

Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. 2013. Extending the Power of Datalog Recursion. 22, 4 (aug 2013), 471–493. https://doi.org/10.1007/s00778-012-0299-1

Jay McCarthy. 2022. *Datalog: Deductive Database Programming.* https://docs.racket-lang.org/datalog/ Accessed 04-13-2023.

Patrick Nappa, David Zhao, Pavle Subotić, and Bernhard Scholz. 2019. Fast parallel equivalence relations in a Datalog compiler. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT).* IEEE, 82–96.

André Pacak, Sebastian Erdweg, and Tamás Szabó. 2020. A Systematic Approach to Deriving Incremental Type Checkers. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 127 (nov 2020), 28 pages. https://doi.org/10.1145/3428195

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

Kenneth A. Ross and Yehoshua Sagiv. 1992. Monotonic Aggregation in Deductive Databases. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (San Diego, California, USA) *(PODS '92).* Association for Computing Machinery, New York, NY, USA, 114–126. https://doi.org/10.1145/137097.137852

Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. In *Proceedings of the 4th International Workshop on the Resurgence of Datalog in Academia and Industry (Datalog-2.0).* Philadelpha, PA, 56–67.

Arash Sahebolamri, Thomas Gilray, and Kristopher Micinski. 2022. Seamless deductive inference via macros. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction.* 77–88.

Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. 2013. Distributed Socialite: A Datalog-Based Language for Large-Scale Graph Analysis. *Proc. VLDB Endow.* 6, 14 (sep 2013), 1906–1917. https://doi.org/10.14778/2556549.2556572

Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big Data Analytics with Datalog Queries on Spark. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16).* Association for Computing Machinery, New York, NY, USA, 1135–1149. https://doi.org/10.1145/2882903.2915229

Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. 2015. Optimizing recursive queries with monotonic aggregates in deals. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 867–878.

Yannis Smaragdakis and Martin Bravenboer. 2011. Using Datalog for Fast and Easy Program Analysis. In *Proceedings of the First International Conference on Datalog Reloaded* (Oxford, UK) *(Datalog'10)*. Springer-Verlag, Berlin, Heidelberg, 245–251. https://doi.org/10.1007/978-3-642-24206-9_14

Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 32–41.

Pavle Subotić, Herbert Jordan, Lijun Chang, Alan Fekete, and Bernhard Scholz. 2018. Automatic Index Selection for Large-scale Datalog Computation. *Proc. VLDB Endow.* 12, 2 (Oct. 2018), 141–153. https://doi.org/10.14778/3282495.3282500

Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. 2021. Incremental Whole-Program Analysis in Datalog with Lattices. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3453483.3454026

Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. (1955).

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915

Rijnard van Tonder. 2021. Towards Fully Declarative Program Analysis via Source Code Transformation. *arXiv preprint arXiv:2112.12398* (2021).

Todd L. Veldhuizen. 2014. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *International Conference on Database Theory*.

Jin Wang, Guorui Xiao, Jiaqi Gu, Jiacheng Wu, and Carlo Zaniolo. 2020. RASQL: A Powerful Language and Its System for Big Data Applications. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2673–2676. https://doi.org/10.1145/3318464.3384677

Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. 2018. RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on a Single Machine. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 763–782.

Aaron Weiss, Olek Gierczak, Daniel Patterson, Nicholas D Matsakis, and Amal Ahmed. 2019. Oxide: The essence of rust. *arXiv preprint arXiv:1903.00982* (2019). https://doi.org/10.48550/arXiv.1903.00982

John Whaley, Dzintars Avots, Michael Carbin, and Monica S Lam. 2005. Using Datalog with binary decision diagrams for program analysis. In *Asian Symposium on Programming Languages and Systems*. Springer, 97–118.

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, Article 23 (jan 2021), 29 pages. https://doi.org/10.1145/3434304

Jiacheng Wu, Jin Wang, and Carlo Zaniolo. 2022. Optimizing parallel recursive datalog evaluation on multicore machines. In *Proceedings of the 2022 International Conference on Management of Data*. 1433–1446.

Carlo Zaniolo, Mohan Yang, Ariyam Das, Alexander Shkapsky, Tyson Condie, and Matteo Interlandi. 2017. Fixpoint semantics and optimization of recursive datalog programs with aggregates. *Theory and Practice of Logic Programming* 17, 5-6 (2017), 1048–1065.

Eric Zhang. 2023. *Datalog compiler embedded in Rust as a procedural macro.* Retrieved March 31, 2023 from 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. arXiv:2304.04332 [cs.PL]

Received 2023-04-14; accepted 2023-08-27