# DESIGNING DATALOG-BASED EMBEDDED LANGUAGES

ERIC K. ZHANG

Integrating logic programming into practical software systems.

March 24, 2023

Eric K. Zhang: *Designing Datalog-Based Embedded Languages,* Integrating logic programming into practical software systems., © March 24, 2023

## ABSTRACT

Datalog is a declarative, domain-specific programming language grounded in principles of logic. At its essence, it expresses computation on relational data in a manner that is both understandable by humans and easy for machines to reason about.

Although it was originally introduced in the 1980s as a subset of Prolog, Datalog has seen a resurgence in a wide variety of modern software applications such as databases, rule engines for security, static analysis frameworks, and authorization. In these cases, the basic semantics of the language are well-established, but its precise syntax and evaluation as a domain-specific language are usually tailored to the application. This opens up exciting possibilities for language design to impact the way people express computational ideas.

To explore the strengths and limitations of Datalog, we design and implement two language systems with Datalog at their core, addressing distinct computer science domains. First, we create a high-performance Datalog implementation in Rust, seamlessly integrate it with the host language, and demonstrate that cross-language function calls are possible. Second, we explore the seldom-examined connection between Datalog and data analysis by creating a reactive web-based notebook programming environment, offering a tangible, reproducible, and easily shareable platform for exploration of datasets.

In both cases, we construct the designed system—either as a prototype or a production-ready library—and present case studies on industry or other real-world usage. Additionally, we benchmark and holistically evaluate the systems to demonstrate how our design choices in embedding Datalog can enhance language interface usability and expressiveness, while also discussing drawbacks and limitations.

*What is software for?*
*People turn to software to learn,*
*to create, and to communicate.*

— Bret Victor [28]

# ACKNOWLEDGMENTS

First and foremost, I'm grateful for my friends and family for their support. Our conversations lent vitality to the past four years; they helped me see the world in vivid color. My work and research interests have been ever-changing. I've often felt personally led to spend weekends and evenings chasing after ideas, a pursuit that would not have been possible without the time we spent together.

The academic direction of this thesis was influenced by several people. Aaron Bembenek was a kind and resourceful guide when I worked with him on Formulog, years ago. I'm grateful to him for encouraging my curiosity as I was introduced to programming languages research.

I'd like to thank Will Crichton for changing the way I view and interact with programming languages, and for pointing me towards interesting literature. I'm also grateful to Kevin Lynagh, Predrag Gruevski, and Pete Vilter for providing valuable feedback on Percival. Jamie Brandon organized the HYTRADBOI conference that allowed me present my work and meet people working at the intersection of databases, programming languages, and interaction design.

I appreciate everyone who contributed code, suggestions, and substantial bug reports to Crepe and Percival: Lara Herzog, Łukasz Jędrzejczyk, Joe Taber, Jake Teton-Landis, McCoy Becker, Remy Wang, J Pratt, and Cornelius Aschermann.

I would like to thank Nada Amin for her serving as my thesis reader and leading seminars on topics in programming langauges at Harvard. I learned a lot from those classes, and their open-ended environment fostered the development of Percival.

Last, my deep gratitude and respect goes to my thesis advisor Steve Chong for welcoming my curiosity and encouraging me to take a step back to think about the bigger picture of research. Steve taught the first computer science class I took at Harvard, and it was very influential.

CONTENTS

# 1 INTRODUCTION

Why develop programming languages? Computers and software are all around us; as of 2021, over 900 Arm microprocessors are manufactured every second [20]. People rely on computers in almost every aspect of their daily lives: learning, communication, creativity, planning, collaboration, and so on. This is only possible if programmers are able to write useful, efficient, and reliable software. To talk to computers, i.e., to express *computational ideas*, we rely on programming languages.

The difficulty in creating good programming languages can be compared to that of any good tool. Languages need to be clear, expressive, and easy-to-use for humans to be productive. We can describe such abstractions as *elegant*, *composable*, *simple*, and *robust*. However, to be useful, languages also need to factor in considerations of the machine: it should be possible to execute code efficiently and reason about it through tooling.

We're lucky in that computational ideas don't just come from thin air. They're timeless, rooted in mathematics. Algorithms invented in 500 BCE work exactly the same way today. So by grounding language design in logic, we can develop sound idioms for interacting with computation.

Different languages draw on various bodies of ideas and work. But the core computational ideas don't change; after all, there are only so many different ways you can tell a person (or machine) to calculate or do something! A programmer or scientist prototyping a simple application who just wants to "get some work done" might write a short, procedural Python script. An engineer building a reliable system for a car or spaceship might use languages that require more planning but offer guarantees around safety and performance. A developer working on geographic information systems might write SQL queries to refine data. These examples all express different slices of ways we can talk to computers, with benefits and tradeoffs to each.

The topic of this thesis is on language design. We explore one compelling avenue towards enriching programming languages with first-class support for querying and inferring results from relational data, and we apply this idea to important problems in multiple disciplines of computer science. By exploring, prototyping, and evaluating our efforts, we can distill ideas that make software easier to write in the future. More fundamentally, advances in language design build toward a unified understanding of how people expressively communicate with computers.

## 1.1 DATALOG

*Datalog historically originated as a subset of a language called Prolog, but this doesn't capture how it is used today.*

Datalog is a declarative logic programming language. Here, *declarative* means that the programmer expresses outcomes rather than the particular steps to get there, and *logic programming* is a paradigm that is based on formal logic. Datalog was first introduced in the 1980s, but it has seen a resurgence in recent years due to applications that benefit from it.

As far as languages go, Datalog is very simple. We will introduce the language in Part I. The basic idea is to specify a form of logical deduction called *rules*, which can possibly refer to each other in a recursive way, to write programs that produce responses to a query. It represents a useful fragment of programming that easily allows people to express queries over structured, relational or graphical data. However, the language itself does not have any standardized syntax and is therefore typically adjusted to its particular problem domain.

One reason why Datalog is interesting is because it is relatively easy to implement, while allowing programmers to very concisely and logically express complex relational analyses, effectively equivalent to SQL with recursive queries. Structured data is everywhere, and the language in its simplicity makes it possible to quickly prototype and add support for such queries without having to implement a complex query planner frontend for alternatives like SQL. It's also more composable in longer programs, and there's substantial literature on efficient evaluation methods.

However, past Datalog implementations have primarily focused on either standalone language systems—interpreters, compilers, or databases—which need to be run separately from application code, or on basic engines that only provide the bare essentials and force the programmer to put the cogs together themselves. This thesis focuses on ways that Datalog can be integrated directly into languages to solve problems.

## 1.2 STRUCTURE

This document is divided into three primary parts, followed by a conclusion. Part I covers background material on Datalog and a brief survey of related work. The next two parts discuss language design for different Datalog applications.

Part II discusses the integration of Datalog as a performant embedded rule engine within Rust, a general-purpose systems programming language. We implement this in a system called Crepe. The unique contribution of this work is that it is the first embedding of Datalog as a procedural macro, allowing for seamless calling of host language functions within the logic programming rules and greatly increasing the speed, ease-of-use, and expressiveness of the language for integrated queries. We

discuss our design and implementation, followed by how Crepe used in the wild by various companies and public open-source projects, and we finish with microbenchmarks and evaluation.

Part III looks at Datalog in the lens of an important but more experimental application: exploratory data analysis. We design and implement Percival, a language and associated reactive notebook environment for data analysis and visualization. In doing so, we make syntactic changes and add support for extra features like aggregations. We also construct a new, compiled Datalog engine that runs as sandboxed JavaScript code within the web browser, allowing people to share notebooks across the web and run them without installing any additional software.

Finally, we synthesize all of the work discussed and make concluding remarks in Part IV.

# Part I

## REVIEW AND BRIEF SURVEY

# 2

# DATALOG SYNTAX AND SEMANTICS

This chapter gives a brief primer into the syntax and semantics of Datalog, as well as common extensions to the basic language. We aim to give a "feel" for the topic by presenting the key ideas, rather than writing a textbook. See [7] for a fuller exposition that complements this chapter.

## 2.1 HORN CLAUSES

The most basic defining element of Datalog as a language is the concept of a *Horn clause*. Horn clauses are logical formulas of the form

$$p \wedge q \wedge \cdots \wedge t \implies u.$$

In other words, some logical conjunction of literals $p, q, \ldots, t$ implies another literal $u$. Horn clauses can be used to do deductive logical inference. For example, suppose that $A(x)$ is the statement that "$x$ is an animal," $L(x, n)$ is the statement that "$x$ has $n$ legs," and $F(x)$ is the statement that "$x$ is a frog." Then, "all frogs are animals" could be written as

$$\forall x : F(x) \implies A(x).$$

Similarly, a statement saying that "all animals with four legs are frogs" would be written as

$$\forall x : A(x) \wedge L(x, 4) \implies F(x).$$

For logic programming, we typically write Horn clauses using a slightly different notation, based on Prolog. Each Horn clause is called a *rule*, which is implicitly universally quantified over all variables in the rule. The rule is written `lhs :- rhs`, where the left-hand side consists of a single *literal*, and the right-hand side has a comma-separated list of literals. For example, the two formulas above could be written in Prolog rules as:

*In other words, the "$\forall x$" part of the formula is assumed.*

```
is_animal(X) :- is_frog(X).
is_frog(X) :- is_animal(X), has_legs(X, 4).
```

You'll notice that our examples so far have been specified to be in Prolog. Why not just say it is Datalog? This is because as a language, Datalog is a subset of Prolog semantically, but its syntax is not well-specified. Different variants of Datalog all have slightly different takes on the syntactical

elements, and there is no one standard. For this chapter, we will give our examples of logic programs in generic Prolog-like syntax until we deal with more specific language systems in later chapters.

With that out of the way, Datalog programs at their core just consist of a list of rules, which are Horn clauses. Each clause connects one or more *relations*, which are mentioned in the clause's literals. There are no builtin mechanisms for input, output, or other side effects. For example, in the program above, the two rules operate on three relations: `is_frog`, `is_animal`, and `has_legs`.

Some of these relations can be populated as *inputs* from a database of facts. These are called the *extensional relations*. A Datalog engine can then run the program, deducing all facts that can be inferred from the input relations, since they appear on the left-hand side of rules. These are called *intensional relations*.

What makes Datalog more interesting is that queries are explicitly allowed to be *recursive* and possibly *mutually recursive*, meaning that they can depend on each other in cycles. This is a key feature. For example, the following Datalog program finds all paths within a directed graph.

```
path(X, Y) :- edge(X, Y).
path(X, Y) :- path(X, Z), edge(Z, Y).
```

In this example, `path` appears on both the left-hand side and the right-hand side of the second rule. This is essential in order for this query to be expressible, since paths in a directed graph can have arbitrary length.

Some logic programming engines generate the total, maximum set of all relations that can be deduced from a database of inputs. Other engines give the user an application programming interface that allows them to query for whether a given relation can be deduced or not. A natural extension of this is to allow the user to provide their own "right-hand side" of rules, whose results are then matched and returned on-demand, as is common in database queries based on Datalog.

## 2.2 FIXED POINT SEMANTICS

What does it really mean to execute a Datalog program? Since they operate on logical predicates, a natural interpretation is that Datalog engines tell you whether a fact can be inferred from its inputs, and if so, which facts those are. Intuitively this makes sense, as the engine is able to automate the "deductive reasoning" part of applying Horn clauses to draw conclusions from relational data, via logic as an interface.

The semantics of Datalog are specified in [4] in detail using three equivalent interpretations, roughly described as model theory, fixed point, and proof theory. The model theory and proof theory interpretations, while

interesting, are not particularly relevant to the rest of this project, so we will skip it but refer the reader to the original source, if they are interested. Instead we discuss its semantics in the fixed point lens.

We will use slightly simpler notation than that of the referenced paper. Let $F$ be the possibly infinite set of all possible facts. The execution of a Datalog program $P$ forms some subset of $F$ containing facts that are derived by the Datalog rules, either directly or recursively. It outputs some member of $\mathcal{P}(F)$, the power set of $F$.

Define a function $f_P : \mathcal{P}(F) \to \mathcal{P}(F)$ that performs one step of deductive inference. In other words, if $S \subset F$ is the set of facts that have been derived thus far, then $f_P(S)$ is a superset of $S$ that contains all facts that are on the left-hand side of a rule (i.e., Horn clause) in the program $P$ currently satisfied by facts in $S$. Equivalently, $f_P$ evaluates one step of all the program's rules at the same time, by instantiating the rules with values for all of their universally quantified variables and deriving new facts using the resulting Horn clauses.

Let $S_0 \subset F$ be the initial set of facts provided as external input to the Datalog program, perhaps from the extensional database (EDB). Then consider the sequence obtained by repeatedly evaluating one step of inference:

$$S_0 \subset f_P(S_0) \subset f_P(f_P(S_0)) \subset f_P(f_P(f_P(S_0))) \subset \cdots,$$

which forms an ascending chain of sets in $\mathcal{P}(F)$. At some point this chain will stabilize at a fixed point, and this *least fixed point* that contains $S_0$ defines the output of the Datalog program.

In plain terms, the program is complete when evaluating a step of deductive inference does not derive any new facts. If a program starts with only ground facts, and there are no functors that operate on the primitive data types, then any produced facts can only involve data from constants present at the start of the program. This implies that $F$ is finite, so the ascending chain must stabilize, and any Datalog engine that continues making progress towards producing facts is guaranteed to terminate.

However, the termination guarantee is not that helpful in practice, as the termination might take polynomial time, where the degree of the polynomial depends on the arity of relations defined in the program. For example, a relation with 26 variables $\text{rel}(A, B, \ldots, Z)$ could take on up to $N^{26}$ values, where $N$ is the number of distinct constants defined at the start of the program. So although mathematically notable, the guaranteed termination claim does not tell programmers that their code will finish executing in any reasonable amount of time. In practice, no Datalog implementation has that property.

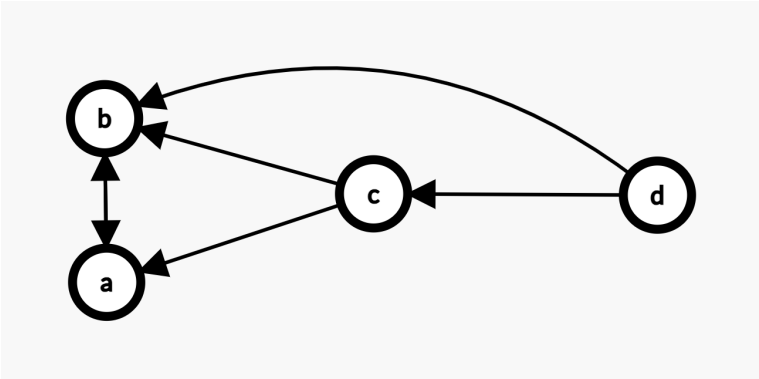



Figure 2.1: Dependency graph in the first Datalog program.

## 2.3 STRATIFICATION

One important property of Datalog programs is that they can be *stratified*, or broken up into one or more parts that have no cyclic dependencies. For example, consider this program.

```
a(X, banana) :- b(X).
b(X) :- a(_, X).

c(X, X) :- b(X), a(X, banana).

d(Z) :- c(X, Z), b(Z).
```

(Here, as a matter of notation, the _ syntax just means any variable that can be ignored and could be replaced with a unique name, and `banana` is a symbol constant, similar to a string in other languages.)

Notice that in this program, relations a and b can be computed first, and all of them could be fully deduced without needing to look at relation c at all. Similarly, all of relation c could be fully computed before looking at any rules that produce relation d. We say that this program can be partitioned into three strata: {a, b}, {c}, and {d}.

If we draw the dependencies between relations in the above program visually, we get the directed graph shown in Fig. 2.1. A directed edge exists from a relation $u$ to a relation $v$ if and only if $v$ appears on the right-hand side of a rule that produces $u$. The strata {a, b}, {c}, and {d} are precisely the strongly-connected components of this graph, and the dependencies can be executed in reverse topological order.

For another illustrative example, consider the trivial but representative Datalog program shown below.

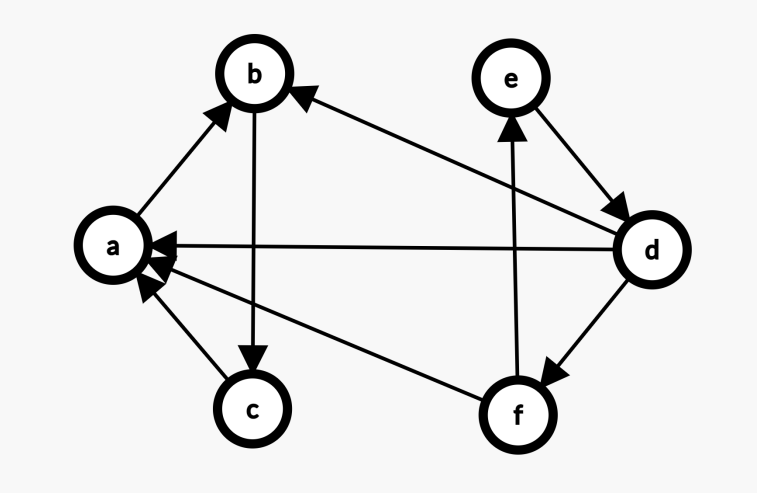



Figure 2.2: Dependency graph in the second Datalog program.

```
a(X) :- b(X). b(X) :- c(X). c(X) :- a(X).
d(X) :- a(X). d(X) :- b(X). d(X) :- f(X).
e(X) :- d(X). f(X) :- e(X). f(X) :- a(X).
```

This program has 6 relations, with a graph of direct dependencies illustrated in Fig. 2.2. Here, we can see that there are directed cycles between each of $\{a, b, c\}$ and $\{d, e, f\}$, so the program can be broken into two strata. The strata match the graph's strongly connected components.

Stratification lets us easily split Datalog programs into multiple sections that can be evaluated independently. This has a couple of advantages:

- Evaluating two sections, one after the other, may be more performant than evaluating both sections in the same pass.
- Stratification allows us to support limited forms of *negation* within Datalog.

What do we mean by negation? A clause in a rule can be *negated* in Datalog, which will only match if that literal cannot be derived by the program. The following shows a simple example, where `!` indicates negation.

*In Prolog terms, negation is satisfied when the program "fails" to derive it.*

```
fruit(mango).
fruit(banana).
fruit(tomato).
vegetable(tomato).
vegetable(cabbage).
output(X) :- fruit(X), !vegetable(X).
```

This program would derive both `output(mango)` and `output(banana)`, but it will not derive `output(tomato)`.

Negation can be allowed only when the program is stratified, meaning that if the negation of a relation $x$ appears on the right-hand side of a rule producing $y$, then $x$ and $y$ must belong to different strata. This allows an engine to hypothetically evaluate it by, for example, fully computing $x$ in an earlier stratum before beginning to evaluate rules with a negated literal, which are non-ambiguous. If negation is not stratified, then a problematic program under the discussed semantics would be be

```
paradox() :- !paradox().
```

If `paradox()` is derived by the program, then it should not be derived, and vice versa. By stratifying negation we avoid this problem.

## 2.4 OTHER EXTENSIONS

Datalog can be extended in various ways that change its semantics. As mentioned previously, every Datalog engine in the wild tends to have its own tailored syntax, features, and therefore semantics. Sometimes features can be quite complex, such as incremental computing, but other times they are simple additions. Not all features are compatible with each other, which makes them interesting to explore.

One difference is in the primitive data types. Prolog originally supports integers, floating-point numbers, atoms (character sequences), and Booleans, but in practice each engine has a different set of supported data types. Any data type can be theoretically supported as long as it is comparable so that rules can perform unification. There are tradeoffs though. Some engines support large data types, which would benefit from reference counting to avoid memory copies, while other engines do not reference count constants for performance reasons.

With data naturally follows the need to perform computations and operations on that data. Adding support for a language of *expressions* to Datalog feels natural but also greatly changes its semantics, as it is no longer guaranteed to terminate. For example, the following program can derive an infinite number of literals: `foo(1)`, `foo(2)`, `foo(3)`, and so on.

```
foo(1).
foo(X + 1) :- foo(X).
```

Arithmetic and string operations still tend to be useful though, as not having them affects the usability of the language. Souffle [10] and Formulog [2], discussed in the next chapter, both support user-defined data types and have a library of of built-in operations, for example. Formulog also allows the user to write their own functions within the language in some fragment of ML-like functional programming.

Besides expressions, another extension is in adding support for *aggregation*. Common aggregation operations like `sum`, `min`, `max`, and `mean` share the common behavior of taking a large slice of data, possibly from a subquery, and computing a single statistic from that data. The Souffle dialect supports aggregation, for example. We will discuss aggregation more in Section 8.2.

Similar to negation, here we will only consider the case where aggregations are stratified. It is possible to handle recursive aggregation operations in some limited scenarios, as described in [24], but this requires using special techniques and assumptions like monotonicity over a lattice to avoid problems analogous to the `paradox()` example above.

# 3 DATALOG EVALUATION METHODS

In the previous chapter we discussed Datalog's syntax and formal interpretations of its semantics. From this we can see that Datalog as a language is very small and easy to understand. However, inside this simple shell is a surprising amount of complexity when it comes to actually evaluating Datalog. Since Datalog supports relational queries with arbitrary recursion, Datalog is able to express a very large number of queries, some of which are difficult to evaluate efficiently.

In a way, Datalog embodies the essence of "relational joins" through a syntax that reminisces equally of logic programming with Horn clauses, or of pattern-matching queries inside a graph database. Depending on extensions added to the language, evaluation methods change as well.

Datalog engines can be standalone, or Datalog can be used as the frontend query language for a database management system. In either case, the evaluation engine needs a way to access the extensional database of base facts and a place to put outputs. Sometimes, the language is implemented in a standalone way with custom, optimized data structures that operate over data files kept in memory.

Within each of these varieties of Datalog runtime, there are different algorithms that can evaluate Datalog with the effective semantics described earlier in Section 2.2. We start by discussing these broad categories of evaluation algorithms.

## 3.1 TOP-DOWN AND BOTTOM-UP

At a high level, the difference between top-down and bottom-up reasoning is between working logically backward and forward. A bottom-up method starts from the set of facts that are known at the beginning, then repeatedly applies deductive rules to expand that set of facts. In contrast, a top-down evaluation method starts from the desired goal themselves and tries to reason backward from the goal, applying rules until it either finds a chain of reasoning that implies that goal, or otherwise fails to find this after exhaustively searching. This involves repeatedly backtracking, rather than gradually growing a set of facts.

By tracing the flow of evaluation, we can check that both of these methods can be implemented to match the semantics of Section 2.2. (A full proof is omitted because the argument is straightforward but notationally cumbersome.) The sketch is as follows. First, either method is able to

prove any fact in the fixed point, since any of those facts must be in some set $f_P(f_P(\cdots f_P(S_0)\cdots))$ and therefore can be deduced by some inference chain, applying Horn clauses in the program. Second, if a fact is not in the fixed point, then there is no chain of rule applications that can deduce that fact, so neither top-down nor bottom-up method can produce it.

Prolog's semantics are defined with depth-first search, a top-down evaluation method, starting from a programmer query. The interpreter executes by repeatedly "unifying" terms with other terms, turning free variables into bound ones. Because Prolog rules can have side-effects like writing to files or taking input, the top-down depth-first search method is an essential part of the language.

On the other hand, Datalog programs do not have side effects, so the engine is free to use any transformations or methods as desired. Most engines evaluate Datalog programs using a bottom-up method. These methods are usually just as effective or better in memory consumption for most programs because in top-down evaluation, an engine must memoize previously-failed evaluations for any nontrivial program. (An example would be graph reachability, where a top-down method would have exponential runtime.) Furthermore, bottomoup methods have advantages for temporal and memory cache locality on CPUs because the same rule can be applied in batches all at once, rather than jumping between rules as needed in an analogous top-down method.

Another reason to prefer bottom-up engines is the existence of program transformation rules that turn top-down programs into equivalent bottom-up programs for Datalog. These are called "magic set" transformations, and they allow a bottom-up program to avoid computing unnecessary facts that are not relevant to deducing the final goal, just as a top-down method would avoid these facts [27]. Several Datalog engines provide implementations in various forms of the magic set transformation.

The simplest bottom-up evaluation method is called *naive evaluation*. It proceeds by closely mimicking Datalog's fix point semantics. First, the engine is started with the set of initial facts $S_0$. On each iteration, the program takes its current database of facts and computes all new facts that can be produced by one step of deductive inference, by iterating over all of the rules and exhaustively unifying them. If any new facts were produced, it then merges those with the previous set and repeats the process with another iteration; otherwise, it terminates.

## 3.2 SEMI-NAIVE EVALUATION

Suppose, as in the last section, that we are doing bottom-up evaluation of a Datalog program. Naive evaluation is fine, but it can often be inefficient. For example in the graph walking program:

```
path(X, Y) :- edge(X, Y).
path(X, Y) :- path(X, Z), edge(Z, Y).
```

When running this in naive evaluation, the set of facts after the $k$-th iteration of rule evaluation would be the pairs of nodes of distance at most $k$ in the graph. Suppose that there are $m$ edges connecting $n$ nodes. Then the maximum number of iterations is $n - 1$, and each naive evaluation iteration requires $O(m)$ time to iterate through potentially all of the edges, so this algorithm has runtime $O(n^2 m)$.

However, the $O(n^2 m)$ time complexity of the naive algorithm is very bad, and we can trivially do a lot better for directed graph reachability by simply running $n$ depth-first search calls in $O(nm)$ time. (Using Tarjan's strongly-connected components algorithm, this specific problem can actually be solved in linear $O(n + m)$ time, but that requires more advanced structural observations specific to the problem. The point here is that even a simple depth-first search outperforms naive evaluation.) What makes naive evaluation so slow is that we need to repeatedly process each of the previous facts on each iteration of rule inference.

Semi-naive evaluation solves this inefficiency by only attempting to evaluate rules where at least one term on the right-hand side of the rule was *generated* (i.e., is new) since the previous iteration. In the case of this graph walk program, on each semi-naive iteration we would only iterate over new paths for the `path(X, Z)` term, rather than all paths for a naive evaluation iteration. This brings the time complexity down for this example down to $O(nm)$.

More generally, semi-naive evaluation can be implemented in an arbitrary stratified program. If a rule has terms $\alpha_1, \ldots, \alpha_k$ that come from the current evaluation stratum, then semi-naive evaluation iterates over, for each $i \in 1, 2, \ldots, k$, the new facts generated for $\alpha_i$ in the past iteration and all facts for terms numbered $\alpha_j$ where $j \neq i$. This is usually more efficient because it reduces the amount of duplicated work that's done in naive evaluation of Datalog programs.

## 3.3 SOURCE CODE TRANSFORMATIONS

Although we've been talking about standalone engines for Datalog so far that deal with primarily in-memory database systems, some Datalog programs are run in other environments. One example is in database systems. Large-scale and distributed databases typically have a query frontend that compiles queries in SQL, Datalog, Cypher, or another query language down into its lower-level intermediate representation, which is then evaluated by the database's storage and executor backend.

Since Datalog is a relational programming language, it can be viewed in a similar way. Datalog programs can be compiled into relational algebra or other specifications before being executed. This is the approach taken several database engines that offer Datalog as one of their supported query languages, such as XTDB and Datomic, as well as tools like Logica and BigDatalog, which both compile Datalog to various query languages running on massive datasets via cloud or cluster-based systems. We will discuss these systems and other related work in Section 4.2.

Finally, closely related to the idea of source code translation is *staging*. We can compile a Datalog program into an efficient executable that runs only that program, which lets us take advantage of low-level compiler optimizations and reduces overhead from parsing and dynamic dispatch. Souffle purports to do this through *Futamura projections*, treating their Datalog interpreter as an object to be specialized on a particular input Datalog program, and the results are positive. This can effectively be viewed as source-to-source translation of Datalog into C++ code. The compiled Datalog systems that we implement and discuss in Parts II and III will use source translation methods with semi-naive evaluation.

# 4

# RELATED WORK

Datalog has proven to be a versatile and flexible language with numerous real-world applications, spanning a wide range of domains from program analysis to big data analytics. In this chapter, we delve into some of the applications of Datalog that are embedded or integrated within larger systems. Although the applications are far too extensive to enumerate comprehensively, we concentrate on some notable examples from various domains, with particular emphasis on those relevant to this thesis.

## 4.1 STATIC ANALYSIS

Datalog has found significant application in the field of static analysis, which involves analyzing source code without executing it. Souffle [10] is a Datalog-based program analysis system that supports efficient and scalable analysis of large codebases. Souffle's distinguishing feature is its high-performance evaluation mechanisms using concurrent B-trees, which work with semi-naive evaluation and a mature compiler to C++ code. Typically, program synthesis in Souffle proceeds by the programmer using facts to represent a codebase, which is then processed by the Datalog engine, allowing for fast and expressive analysis of code.

Formulog [2] is another Datalog-based system that combines logic programming with Satisfiability Modulo Theories (SMT) solvers and a first-order subset of ML (a functional programming language) to enhance the performance of static analysis. It leverages the strengths of both SMT solvers and Datalog by embedding satisfiability calls and model checking within rules, thereby allowing the system to reason about more complex program properties. Formulog has been used to implement systems like a dependent type checker, a symbolic execution engine, and standard points-to and escape analyses.

CodeQL [6] is a commercial and open-source structural query language for code that is used to find security vulnerabilities. It is semantically based on Datalog and is widely used in industry. CodeQL allows users to write Datalog-based queries that analyze code for security vulnerabilities, including SQL injection, cross-site scripting, and buffer overflow vulnerabilities.

Ascent [19] is an embedded Datalog engine that is very similar in design to Crepe, the system we create and evaluate in Part II of this thesis. Work on the Ascent project began more than 12 months after Crepe was first

published as a library and widely distributed in the open-source Rust community, being downloaded thousands of times and used by several projects. The authors of Ascent have seen Crepe, but they failed to cite it as prior work in their conference publication.

## 4.2 DATABASE QUERY GRAMMARS

Datalog has been used as a database query language in several systems, where its simplicity and ability to express recursive logic allow users to query data effectively. Datalog queries are often simpler and more intuitive than SQL queries in these databases, and they can be used to express complex queries that are difficult to express using SQL.

Datomic is a distributed graph database system that allows for flexible schema changes and provides ACID transactions. It is based on the principles of Datalog, but extends it to handle transactions. A dialect of Datalog is used to express the queries, and Datomic's query engine is optimized for efficient execution on auto-scaling distributed clusters. It also provides a rich set of query primitives, including aggregations, filtering, sorting, and joins. Datomic is used in various domains, including finance, healthcare, and e-commerce.

DataScript is an in-memory embedded database with a Datalog query engine. It is designed for Clojure and ClojureScript applications. DataScript uses a syntax compatible with Datomic and therefore supports recursive queries, joins, and aggregations, and provides a set of built-in functions for manipulating data. It also allows for easy integration with other Clojure libraries and frameworks. Other projects like Datalog UI have been drawn inspiration from DataScript's use of Datalog as an embedded application database.

Logica [23] is a Datalog compiler that translates Datalog queries into StandardSQL for execution on Google BigQuery. This allows the use of Datalog in the context of very large datasets. Logica was developed by Google, and it allows programmers to write queries that are more composable, reusable, and understandable than if they had been developed directly in SQL.

BigDatalog [22] is a Datalog-based big data analytics system that runs on Apache Spark. It supports the execution of complex Datalog queries on large-scale data. BigDatalog provides a high-level query language that allows users to write Datalog queries that are automatically translated into Spark code for execution. Other distributed Datalog implementations that are comparable include SociaLite [21] and Myria [29].

## 4.3 AUTHENTICATION AND ACCESS CONTROL

Datalog has also been employed for authorization and role management in enterprise software systems. One such example is Oso Cloud, a cloud-based authorization system that leverages Datalog to define access policies and permissions. By utilizing Datalog, Oso Cloud can create expressive and flexible rules for managing access rights, enabling a high degree of customization and adaptability to various use cases.

DDlog [18], a dialect of Datalog developed at VMware, supports incremental computation based on differential dataflow. This approach has been employed for network control applications, encompassing packet processing and software-defined networking (see Nerpa). The use of DDlog enables efficient and responsive computation of network state changes, making it suitable for applications that demand real-time updates and rapid adaptation to evolving conditions.

## 4.4 AI KNOWLEDGE ENGINES

Datalog has been utilized as a knowledge representation language in AI knowledge engines, demonstrating its efficacy in representing and reasoning about complex data relationships. LogicBlox [1] is a commercial knowledge graph system that employs Datalog to facilitate complex reasoning and inferencing tasks. By adopting Datalog as its underlying language, LogicBlox can manage intricate dependencies and interconnections between entities in the knowledge graph, enabling powerful queries and analysis. RelationalAI is the successor to LogicBlox, also used as a knowledge graph system that uses Datalog to support reasoning tasks.

Part II

# DATALOG EMBEDDED AS A RELATIONAL ENGINE

# 5

# SEAMLESSLY COMPILING LOGIC PROGRAMS IN RUST

The applications of Datalog described in the last section were mainly standalone systems that could communicate with other software over the network as a client-server protocol, or as a subprocess running on database files. However, there's equally a lot of creative potential in using Datalog as a relational core that can be embedded in larger programs.

This is difficult to do in most high-performance Datalog systems. For example, in Souffle, to embed Datalog in a larger program, you would need to pre-compile the Souffle code into a native executable, package that executable with your program for all architectures, create a temporary file with your input, spawn Souffle as a subprocess, and read the output from another file. This procedure is tedious and error-prone, and there are many places for hard-to-detect failures to arise, such as if the runtime ran out of memory, was killed by the operating system, or froze due to a deadlock. It also creates a great deal of operational dependency between different software components.

For Datalog to be effectively used as the logical engine within larger systems software, such as compilers or control mechanisms, it needs to be possible to execute relational code from within larger programs. In other words, the Datalog engine should run as a data structure within a *library*, rather than a standalone engine.

Libraries have their own benefits and drawbacks compared to standalone software though. The principal advantages are that they are simpler to use and integrate with inside the host language, and they can provide much more expressive cross-language interaction compared to an interprocess communication or network-based interface. For example, library APIs can take first-class function callbacks as arguments. They also tend to have less overhead in serializing data structures between languages.

The downsides unfortunately are that libraries typically are restrained in syntax since they have to parse within the host language, which can be awkward or clumsy. They also are not usable from different languages other than the one that they are written for, and since there is less processing that can be done at compilation time, it may lead to slower code.

Is it possible to use Datalog as a *library* without sacrificing performance or expressiveness? In this part I introduce Crepe, a high-performance compiled Datalog engine embedded in Rust as a procedural macro that integrates seamlessly with Rust code. This is the first library (though others have since followed) that embeds Datalog within a general-purpose host

language while offering cross-language function calls. I demonstrate that this embedded language is efficient and expressive, and I show real-world examples of deductive programs using Crepe written by companies in production, and by research projects.

## 5.1 EMBEDDED LANGUAGE SYSTEMS

Let's give concrete examples of how embedding languages as a library differs from other mechanisms. The key difference is in the application interface. An algorithm does not have knowledge of the interface by which it is presented to the world, but this interface can greatly affect its performance, usability, and expressiveness when presented to users.

We can see fitting examples of this tradeoff in databases and database query languages. For example, common relational databases run with a client-server model, one example being PostgreSQL [16], which allows query engines from many different applications to interact with the software while being optimized as a system running on one or more nodes that are designed to manage the database. The interface presented is in its own DSL, usually SQL, which is used the same way from many languages. That said, people often use libraries like object-relational mappers to interact with these databases as a result, which creates inefficiencies because these libraries need to convert between the database's communication and language-specific types, which don't always match up [9]. If the database is served over the network, extra back-and-forth requests also increase latency and add wire overhead.

*Another common problem is native data type mismatch: for example, the popular Firebase database silently strips bits off 64-bit integers when used from JavaScript.*

Embedded database engines like SQLite are able to sidestep some of these issue by compiling the database within the program itself, which allows all of the data to live within the same process and eliminates communication overhead between the engine and application [8]. Furthermore, popular data analysis libraries like Pandas [13] can be alternatively viewed as in-memory data query systems that avoid the object-relational mismatch entirely by presenting their own, succinct APIs that operate on language-specific data types. For example, compared the following queries using a PostgreSQL database and Pandas from Python:

```
# Using SQL
import psycopg

with psycopg.connect(database_url) as conn:
    with conn.cursor() as cur:
        cur.execute("""
SELECT state, COUNT(*) FROM people
WHERE name LIKE 'Eric %'
```

```
GROUP BY state;
""")
        results = cur.fetchall()

# Using Pandas
import pandas as pd

df = pd.read_csv("people.csv")
results = (
    df.loc[df.name.str.startswith("Eric ")]
      .groupby("state").count()
)
```

The latter query works strictly within Python types, while the first query needs to manage resources like connections and cursors. It's also written in a separate language within a string expression that cannot be checked for syntactical accuracy. Although one could in theory use a PostgreSQL connector like `psycopg` with data that's loaded in memory, in practice embedded language-specific query engines like `pandas` are much more popular because they are faster, easier to use, and more familiar to programmers, as ordinary Python code.

We've drawn a parallel to embedding languages in traditional database engines, but the idea of embedding libraries is common in many different software applications. There are often idiomatic *library bindings* written for software in specific languages: just a few examples are Z3's `z3py` for Python, the `nix` crate wrapping libc for Rust, and the `Stripe.js` library for Stripe's HTTP API in JavaScript. Even Souffle has experimental support for Haskell via `import Language.Souffle.Compiled`. But these libraries, although they make interacting with the software easier within a specific language, can't make the interface more powerful or expressive. for example, you will never be able to inject Python code into a Z3 predicate because it is not representable within the intermediate format.

In summary, embedding languages provides a powerful interface that "over the wire" systems cannot provide, simply because they are separated from the main program using them. These differences affect how convenient it is to use languages, and programmers tend to prefer libraries over bindings.

## 5.2 PROCEDURAL MACROS

In order to get the benefits of Datalog's concise syntax while also being embedded as a library and taking advantage of compiler optimizations, we implement Crepe using *procedural macros*.

A procedural macro is effectively a function that acts as a translator or basic compiler. It parses some grammar and produces source code from it, which is then inserted into the program and compiled with the rest of it. This is typically used to generate repetitive code. For example, one simple macro from [26] is called `seq!`, and it is used like so:

```
seq!(N in 0..512 {
    #[derive(Copy, Clone, PartialEq, Debug)]
    pub enum Processor {
        #(
            Cpu~N,
        )*
    }
});
```

This module is a very short code snippet, and it includes some fragments of syntax like `N in 0..512 {..}` and `#( Cpu=N, )*` that would ordinarily not be valid Rust code. However, the `seq!()` macro is able to intercept compilation and transform the code into the equivalent:

```
#[derive(Copy, Clone, PartialEq, Debug)]
pub enum Processor {
    Cpu0,
    Cpu1,
    Cpu2,
    Cpu3,
    Cpu4,
    // ... 505 lines removed for brevity
    Cpu510,
    Cpu511,
}
```

The process of replacing macros by the source code they generate is called *macro expansion*. The compiler executes code written in a separate package that conforms to a specific API. For example, `seq!` is implemented using a function like:

```
#[proc_macro]
pub fn seq(input: TokenStream) -> TokenStream {
    // ... transform the input code to output code
}
```

This function has full access to the token stream and can essentially run any procedure on it. Here, the procedure just looks for the tokens `N in 0..512` and searches for fragments of expressions wrapped by `#()*`, replacing all occurrences of `N` in their contents with each number in the integer

range from 0 to 512. However, the developer has full freedom to interpret arbitrary syntax in procedural macros as they wish, so although distasteful, one could just as easily require the macro to be used like:

```
seq!(hello, do this for every N between ((0 and 512)) {
    #[derive(Copy, Clone, PartialEq, Debug)]
    pub enum Processor {
        (START REPEAT)
            Cpu~{{N}},
        (END REPEAT)
    }
});
```

Procedural macros are a core part of Rust as a language, and libraries like `serde` [25] that rely on procedural macros for code generation have been downloaded over 140 million times. They are not present in many other languages like C++, Python, and Java, although this isn't to say that they are unique to Rust, as other languages like Julia, Nim, and Lisp have similar features. (C and C++ do have simple preprocessor directives, but they do not allow the programmer to run code.)

Crepe takes advantage of procedural macros to their fullest extent: not just as generators of repetitive code, but as full compilers that transform arbitrary language syntax into Rust. Similar to how Souffle generates efficient C++ code from a separate Datalog program before compile time, Crepe generates efficient Rust code from a Datalog program. The difference is that Crepe runs *during compile-time* using procedural macros and can be more easily embedded into larger programs, as we will see.

## 5.3 COMPILING DATALOG TO RUST CODE

Now that we described procedural macros in Rust, we can share how Crepe works. The most basic example of a Datalog program in Crepe is the following:

*Notes about Rust: the type **`i32`** represents a 32-bit signed integer, and **`struct`** is a feature that defines new record data types.*

```
use crepe::crepe;

crepe! {
    @input
    struct Edge(i32, i32);

    @output
    struct Reachable(i32, i32);

    Reachable(x, y) <- Edge(x, y);
```

```
    Reachable(x, z) <- Edge(x, y), Reachable(y, z);
}

fn main() {
    let mut runtime = Crepe::new();
    runtime.extend([Edge(1, 2), Edge(2, 3), Edge(2, 5)]);

    let (reachable,) = runtime.run();
    for Reachable(x, y) in reachable {
        println!("node {} can reach node {}", x, y);
    }
}
```

This produces five lines of output, in some order:

```
node 1 can reach node 2
node 1 can reach node 3
node 1 can reach node 5
node 2 can reach node 3
node 2 can reach node 5
```

The code within `crepe! { .. }` is processed by Crepe, and the rest of the code in the `main` function shows how you can call Crepe from outside code. This program takes a directed graph as input and computes its *transitive closure*, i.e., all pairs of nodes $(x, y)$ such that there exists a directed path in the graph from $x$ to $y$.

The code generated by the `crepe!` macro includes the definition of a `Crepe` "struct" in Rust, which is a template for an object. The object has a few methods for interacting with the Datalog runtime:

- `let mut runtime = Crepe::new()` constructs a new runtime, and the `mut` keyword in Rust declares the variable as mutable.
- `runtime.extend([..])` extends the runtime with input relations (marked by `@input`) from an iterable collection.
- `runtime.run()` consumes the runtime, running the compiled Datalog program and eventually returning all generated outputs (marked by `@output`) as sets.

Another way of describing this is that the interface has roughly the following shape, in Rust pseudocode:

```
use std::collections::HashSet;

/// Object representing the Crepe runtime.
```

```rust
pub struct Crepe { .. }

impl Crepe {
    /// Construct a new Datalog engine.
    pub fn new() -> Self {
        ..
    }

    /// Run the Datalog program, returning all output relations.
    pub fn run(self) -> (HashSet<Reachable>,) {
        ..
    }
}

impl Extend<Edge> for Crepe {
    /// Add one or more edges to the engine as inputs.
    fn extend(&mut self, iter: impl IntoIterator<Item = Edge> {
        ..
    }
}
```

This is the most basic example of usage. Crepe has more features and also supports things like stratified negation, calling Rust functions from directly within Datalog rules, and extensions for disaggregation and matching with refutable patterns. But before getting to that, we first discuss how Crepe compiles this Datalog program to Rust code.

### 5.3.1 *Naive evaluation*

The simplest evaluation method is naive evaluation, which repeatedly iterates over the set of relations and generates new ones until reaching a fix point, as discussed in Section 3.1. We can implement this in Crepe by generating source code that corresponds to multiple nested `for`-loops.

Let's walk through what this looks like for the transitive closure example. The function containing all of the relevant code is `Crepe::run()`, since that actually evaluates a Datalog program. First, define the relation sets:

```rust
let mut __edge: HashSet<Edge> = HashSet::new();
let mut __edge_update: HashSet<Edge> = HashSet::new();
let mut __reachable: HashSet<Reachable> = HashSet::new();
let mut __reachable_update: HashSet<Reachable> = HashSet::new();
```

We also define an index called `__reachable_index_bf` for the reachable relation, whose utility will be apparent soon. This is defined so that the

entry corresponding to an integer `x` is the set of all tuples of the form `Reachable(x, _)`. The `_bf` in its name comes from the first variable being *bound*, while the second variable is *free*.

```
let mut __reachable_index_bf: HashMap<(i32,), Vec<Reachable>> =
    HashMap::default();
```

To initialize our runtime, we pass the `@input` data into the state. Assume that this is stored in variable `self.edge` in the runtime.

```
__edge_update.extend(self.edge);
```

Now, we can start running the main loop that evaluates rules. We need to run this loop at least once, but after the first iteration, if all of the update sets are empty, we've reached a fix point and can stop. The first thing we'll do in the loop is update all sets with the new facts generated in the previous iteration.

```
let mut __crepe_first_iteration = true;
while __crepe_first_iteration ||
    !(__edge_update.is_empty() && __reachable_update.is_empty()) {
    __reachable.extend(&__reachable_update);
    for __crepe_var in __reachable_update.iter() {
        __reachable_index_bf
            .entry((__crepe_var.0,))
            .or_default()
            .push(*__crepe_var);
    }
    __edge.extend(&__edge_update);
```

Next, we define two `_new` sets within the inner loop, which will store the new facts generated during this iteration. We'll set the `_update` sets to these at the end of the loop body.

```
    let mut __edge_new: HashSet<Edge> = HashSet::new();
    let mut __reachable_new: HashSet<Reachable> = HashSet::new();
```

With all of this in place, it's finally time to generate code for the rules. We compile the first rule, `Reachable(x, y) <- Edge(x, y)` into Rust code.

```
    for &Edge(x, y) in &__edge {
        let __crepe_goal = Reachable(x, y);
        if !__reachable.contains(&__crepe_goal) {
            __reachable_new.insert(__crepe_goal);
        }
    }
```

The second rule `Reachable(x, z) <- Edge(x, y), Reachable(y, z)` is similar, except it involves two nested loops because there are two clauses. Note that we use the index we created earlier to avoid having to search through unnecessary data.

```
    for &Edge(x, y) in &__edge {
        if let Some(__crepe_iter) = __reachable_index_bf.get(&(y,)) {
            for &Edge(_, z) in __crepe_iter {
                let __crepe_goal = Reachable(x, z);
                if !__reachable.contains(&__crepe_goal) {
                    __reachable_new.insert(__crepe_goal);
                }
            }
        }
    }
```

Finally, we finish the main loop.

```
    __reachable_update = __reachable_new;
    __edge_update = __edge_new;
    __crepe_first_iteration = false;
}
```

That's it! When the loop terminates, `__reachable` will contain the full set of reachable pairs computed by this Datalog program. We have translated naive, bottom-up evaluation into Rust code in a mechanical way.

### 5.3.2 *Semi-naive evaluation*

Naive evaluation is not very efficient, and we can do a lot better by only consider running the rules on incremental deltas from the previous iteration. This was described in Section 3.2. Luckily, extending the above system to support semi-naive evaluation is very simple. We only have to make a few minor changes to the code.

First, at the start of the evaluation function, we define an additional index that stores only facts in `__reachable_update`.

```
let mut __reachable_index_bf_update: HashMap<(i32,), Vec<Reachable>> =
    HashMap::default();
```

We modify the code to do bookkeeping on `__reachable_index_bf_update`, omitted for brevity. The more interesting part is how rules are compiled differently. The first rule is almost the same but reads from `__edge_update` instead of `__edge`, since it doesn't need to ever look at old edges twice.

```
for &Edge(x, y) in &__edge_update {
    let __crepe_goal = Reachable(x, y);
    if !__reachable.contains(&__crepe_goal) {
        __reachable_new.insert(__crepe_goal);
    }
}
```

The second rule has two clauses, so it will be compiled into two evaluation loops.

```
for &Edge(x, y) in &__edge_update {
    if let Some(__crepe_iter) = __reachable_index_bf.get(&(y,)) {
        for &Edge(_, z) in __crepe_iter {
            let __crepe_goal = Reachable(x, z);
            if !__reachable.contains(&__crepe_goal) {
                __reachable_new.insert(__crepe_goal);
            }
        }
    }
}

for &Edge(x, y) in &__edge {
    if let Some(__crepe_iter) = __reachable_index_bf_update.get(&(y,)) {
        for &Edge(_, z) in __crepe_iter {
            let __crepe_goal = Reachable(x, z);
            if !__reachable.contains(&__crepe_goal) {
                __reachable_new.insert(__crepe_goal);
            }
        }
    }
}
```

In general, if a rule has $n$ clauses consisting of different relations, then the semi-naive evaluation strategy will have $n$ evaluation loops. The $i$-th loop will check over only the update set for the relation in index $i$ and loop over the full sets in all other indices.

### 5.3.3 *Stratification*

The last few subsections have shown how to translate a Datalog program into compiled Rust code that generates a runtime and evaluates it. However, many extensions to Datalog require stratification to work properly, such as negation. Crepe supports stratified negation, so it needs to be able to split evaluation of programs into strata.

It does this by breaking up the main loop into one outer loop for every stratum. These are determined by the strongly connected components of the relation dependency graph, where each relation $r_1$ has a directed edge pointing toward $r_2$ if $r_1$, or its negation, appears on the right-hand side of some Horn clause producing $r_2$.

Strongly connected components can be computed in linear time with well-known graph algorithms, so we are done. For example, this program would compile in Crepe and compute the difference of two sets:

```
crepe! {
    @input
    struct A(i32);

    @input
    struct B(i32);

    @output
    struct C(i32);

    C(x) <- A(x), !B(x);
}
```

We refuse to compile programs that have the negation of $r_1$ on the right-hand side of a Horn clause producing $r_2$ if $r_1, r_2$ belong to the same strongly-connected component in this graph, as they are not well-formed. Here is an example of a program that doesn't work:

```
crepe! {
    @input
    struct A(i32);

    @output
    struct B(i32);

    B(x) <- A(x), !B(x);
}
```

This program is not well-defined, since the fact $B(x)$ is implied by the negation of $B(x)$, which leads to a contradiction. Crepe indicates this by presenting the user with a compiler error.

```
error: Negation of relation 'B' creates a dependency cycle and cannot be stratified.
 --> $DIR/recursive_negation.rs:8:14
   |
11 |     B(x) <- A(x), !B(x);
   |                   ^
```

## 5.4 HOST LANGUAGE INTEGRATION

Compiling Datalog directly to Rust code within a procedural macro lets us take advantage of Rust's compiler optimizations, as well as LLVM passes, which combine to generate high-performance code. However, it's still a Datalog program without any specific syntax other than data types.

This section outlines additional syntax extensions that allow the user to seamlessly integrate Crepe programs in Rust, specifically by including raw code expressions and outside logic. These also let us borrow constants, functions, and user-defined types from the surrounding Rust code.

### 5.4.1 *Conditionals and expressions*

Crepe supports arbitrary Rust expression syntax within rules for constructing new relations, i.e., on the left-hand side of Horn clauses. It also allows the user to write Boolean expressions evaluated directly as conditional clauses in rules, if they are surrounded by parentheses.

As an example for demonstration, here is a program that calculates the first 25 Fibonacci numbers using Rust's arithmetic and comparison operators.

```
crepe! {
    @output
    struct Fib(u32, u32);

    Fib(0, 0) <- (true);
    Fib(1, 1); // shorthand for `Fib(1, 1) <- (true);`

    Fib(n + 2, x + y) <- Fib(n, x), Fib(n + 1, y), (n + 2 <= 25);
}
```

Let $F_n$ refer to the $n$-th Fiboacci number. This program first populates the `Fib` relation with two facts, $\mathrm{Fib}(0, 0)$ and $\mathrm{Fib}(1, 1)$. The next rule then takes $\mathrm{Fib}(n, F_n)$ and $\mathrm{Fib}(n+1, F_{n+1})$, checks if $n + 2 \leqslant 25$, and deduces $\mathrm{Fib}(n + 2, F_n + F_{n+1})$ if so. It terminates when no more relations can be generated, computing the values of $F_0$ through $F_{25}$.

The Fibonacci example is not practical compared to just writing a typical imperative loop in Rust, but we will see real-world uses of conditionals and expressions in Chapter 6.

Crepe also supports variable "`let`"-bindings in rules, including bindings that match their arguments conditionally against a pattern. Here is an example that uses the fallible `str::parse(&self)` method from the Rust standard library. Here, the expression `string.parse()` tries to parse an integer from a string and returns `Ok(n)` only if it succeeded.

```
crepe! {
    @input
    struct Value<'a>(&'a str);

    @output
    struct Squared(i32, i32);

    Squared(n, x) <-
        Value(string),
        let Ok(n) = string.parse(),
        let x = n * n;
}
```

For example, if the inputs were `Value("-3")`, `Value("12")`, and finally `Value("hello")`, then the two outputs would be `Squared(-2, 4)` and `Squared(12, 144)`.

Conditional destructuring is particularly useful for logical reasoning on program data, which often involves algebraic data types (or **enum**s in Rust). Since Crepe is a macro, it can use user-defined types from the surrounding scope, as below.

```
#[derive(Copy, Clone, Hash, Eq, PartialEq)]
enum Token {
    String(&'static str),
    Integer(i32),
    Fraction(i32, i32),
}

crepe! {
    @input
    struct ProgramToken(Token);

    @output
    struct ProgramString(&'static str);

    @output
    struct ProgramInteger(i32);

    ProgramString(s) <-
        ProgramToken(t),
        let Token::String(s) = t;

    ProgramInteger(x) <-
        ProgramToken(t),
```

```
        let Token::Integer(x) = t;

    ProgramInteger(q) <-
        ProgramToken(t),
        let Token::Fraction(x, y) = t,
        (y != 0 && x % y == 0),
        let q = x / y;
}
```

This program takes a collection of `Token`s as input and destructures each of them based on their variant to generate specific output facts. The facts could then be used as intermediates in a larger program analysis.

The last built-in control flow feature is iteration over data. Rules can enumerate values from an iterator, allowing them to use data from outside of Crepe without having to convert functions to use workarounds. For example, to access the characters of a string, you could write:

```
crepe! {
    @input
    struct Name<'a>(&'a str);

    @output
    struct NameContainsLetter<'a>(&'a str, char);

    NameContainsLetter(name, letter) <-
        Name(name),
        for letter in name.chars();
}
```

Iteration over data can be combined with lazy evaluation. For example, in a graph analysis with multiple nodes, a function could be used to lazily generate the edges adjacent to each node in a graph analysis as they are visited for the first time. Because Crepe uses semi-naive evaluation, these functions can be evaluated only once if they are expensive, during the first iteration that each node is added to the set of relations.

### 5.4.2 *Convergence properties*

These control flow primitives are very powerful, but they do come with some responsibility. For example, Datalog is guaranteed to terminate, and there is no undefined behavior or side effects. Rust is not guaranteed to terminate, and any outside functions called from Crepe rules may have arbitrary side effects, ranging from benign printing and logging to complex operations like HTTP queries and reading from files. They also might not

be deterministic; one could imagine calling a function that generates a random number.

```
crepe! {
    @output
    Rel(x) <- let x = fastrand::f64();
}
```

Or a function that returns a different value each time it is executed (although this is unnatural to do in Rust due to its borrow checker):

```
use std::sync::atomic::{AtomicU32, Ordering};

static GLOBAL_VAR: AtomicU32 = AtomicU32::new(0);

fn some_number() -> u32 {
    GLOBAL_VAR.fetch_add(1, Ordering::Relaxed)
}

crepe! {
    @input
    struct Foo(&'static str);

    @output
    struct Bar(&'static str, u32);

    Bar(s, some_number()) <- Foo(s);
}
```

Examples like these break the purity guarantee of Datalog. However, as long as user-defined functions are written to be pure expressions of the inputs, then execution of Crepe will still be deterministic.

Programs also need to be written to avoid an infinite set of relations being generated, as then they will not terminate. Therefore, programs need not always converge or even produce the same output in pathological scenarios, but this is a tradeoff that gives Crepe extra flexibility by being tightly integrated with its host language.

# 6 CASE STUDIES ON PUBLIC USAGE OF CREPE

Crepe was publicly released in September 2020. In the following years, there has been public usage of Crepe by various individuals, companies, and open-source projects. As of March 2023, it has been downloaded 105,000 times as a Rust library on crates.io. In this chapter we briefly illustrate how people use Crepe's features in real-world scenarios.

## 6.1 ACCESS CONTROL FOR CLOUD INFRASTRUCTURE

Teleport is a software-as-a-service company that creates and offers hosting for cloud infrastructure systems for identity, access management, and security. Traditionally these are difficult-to-use and complex, so they have released open-source software that can be deployed in cloud environments to manage secrets, authorization, and infrastructure access for developers. It runs as a user binary on Linux, Kubernetes, and database clusters. According to their website, Teleport is deployed by companies including DoorDash, Elastic, Vonage, Nasdaq, and Snowflake.

Most of Teleport's open source code is written in Go, C, and TypeScript, with Rust only making up a small fraction of its codebase. Its core authorization engine that tests if users can login to access a certain node on its network is specified using Datalog. The code in v9.3.26 of the software uses Crepe to test for role access, using bottom-up evaluation:

```
const LOGIN_TRAIT_HASH: u32 = 0;

crepe! {
    // Input from EDB
    @input
    struct HasRole(u32, u32);
    @input
    struct HasTrait(u32, u32, u32);
    @input
    struct NodeHasLabel(u32, u32, u32);
    // ... for brevity, 8 more lines omitted

    // Intermediate rules
    struct HasAllowNodeLabel(u32, u32, u32, u32);
    struct HasDenyNodeLabel(u32, u32, u32, u32);
```

```
    // ... for brevity, 3 more lines omitted

    // Output for IDB
    @output
    struct HasAccess(u32, u32, u32, u32);
    @output
    struct DenyAccess(u32, u32, u32, u32);
    @output
    struct DenyLogins(u32, u32, u32);

    // Intermediate rules to help determine access
    HasAllowNodeLabel(role, node, key, value) <-
        RoleAllowsNodeLabel(role, key, value),
        NodeHasLabel(node, key, value);
    HasDenyNodeLabel(role, node, key, value) <-
        RoleDeniesNodeLabel(role, key, value),
        NodeHasLabel(node, key, value);
    HasAllowRole(user, login, node, role) <-
        HasRole(user, role),
        HasAllowNodeLabel(role, node, _, _),
        RoleAllowsLogin(role, login),
        !RoleDeniesLogin(role, login);
    // ... for brevity, 16 more lines omitted

    // HasAccess rule determines each access for a specified
    // user, login and node
    HasAccess(user, login, node, role) <-
        HasAllowRole(user, login, node, role),
        !HasDenyRole(user, node, _),
        !HasDeniedLogin(user, login, _);
    DenyAccess(user, login, node, role) <-
        HasDenyRole(user, node, role),
        HasTrait(user, LOGIN_TRAIT_HASH, login);
    DenyAccess(user, login, node, role) <-
        HasDenyRole(user, node, role),
        HasAllowRole(user, login, node, _);

    DenyLogins(user, login, role) <-
        HasDeniedLogin(user, login, role);
}
```

This snippet is fairly long, but every line in the Crepe macro clearly describes a logical rule that forms part of their access control system

in Datalog. This allows the code to use Crepe as a succinct embedded language to quickly implement a role-based matching engine that joins together seven relations of input data.

Embedding in this case also makes the engine faster. As mentioned previously, Teleport's codebase is primarily Go. Even though Crepe is written and has a programming interface in Rust code, this codebase was able to limit its Rust usage to a single package for the role tester and compile that to a shared library, which could then be used via foreign function interface in Go. Therefore, the role tester remains embedded and does not have to go through the overhead of launching a separate subprocess, or communicating through network sockets, to compute the result of an access query.

## 6.2 JOB SCHEDULING IN A STREAMING SQL DATABASE

Another project that uses Crepe is RisingWave, a distributed SQL database for stream processing. This is an open-source database systems project, which is used in production by "dozens of companies across a diverse range of industries, including entertainment, fintech, social media, and manufacturing." It has been compared (see [31]) to other incremental query processing systems like Materialize, which is based on differential dataflow [15].

The database acts as a distributed incremental query engine, accepting data from streams like Apache Kafka and other similar sources to incrementally maintain materialized views, which it uses to respond to queries from a PostgreSQL-compatible wire protocol. The incremental maintenance allows it to respond to queries in real time.

The system is written entirely in Rust, which makes it a natural fit for Crepe's embedded language API. RisingWave v0.1.17 uses Crepe in its distributed scheduler for queries across multiple database shards. This scheduler operates on a query graph and performs a dataflow analysis while resolving the query, also checking for conflicts.

The change that replaced their previous, homegrown scheduler with a new version based on Crepe eliminated over 500 lines of code. In the words of the author in their pull request, it also helped "simplify" and "generalize" their logic, "make the whole [scheduler] progress much clearer," and "improve the readability a lot."

Here are the core snippets of code that implement their stream graph scheduler. It includes a sampling of many of Crepe's features mentioned in Chapter 5 as an embedded language. First, they define several data types in ordinary Rust syntax. (Comments in code quoted in this section have been lightly edited for clarity.)

```rust
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
struct Id(u32); // NB: this type is simplified for exposition

type ParallelUnitId = u32;
type HashMappingId = usize;

/// Distribution ID processed in the scheduler.
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
enum DistId {
    Singleton(ParallelUnitId),
    Hash(HashMappingId),
}

/// Facts as the input of the scheduler.
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
enum Fact {
    /// An edge in the stream graph.
    Edge {
        from: Id,
        to: Id,
        dt: DispatcherType,
    },
    /// A distribution requirement for an external fragment.
    ExternalReq { id: Id, dist: DistId },
    /// A singleton requirement for a building fragment.
    SingletonReq(Id),
}

/// Results of all building fragments, as the output of the scheduler.
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
enum Result {
    // ... omitted for brevity
}
```

The user-defined types in this snippet, such as the `ParallelUnitId` type alias and the `DistId` tagged union type just use plain Rust features. All of this code so far could have been written naturally without Crepe, but Crepe can process these user-defined types regardless.

The big picture is that the `Fact` enum defines certain dependencies between nodes with integer identifiers, which represent a directed graph. Then, the scheduler returns a set of `Results`, which determine where different fragments of the query are executed. The remaining details of the scheduler are not important for this discussion.

Immediately following these data type definitions follows the invocation of the `crepe!` macro, which performs a dataflow analysis with Datalog. In a total of just 13 rules, this logic program defines the scheduling system for a stream graph running on distributed database workers, including logic for detecting failures. It weaves together a combination of features like algebraic data types, stratified negation, and embedded Rust expressions to work naturally with the surrounding code.

```
crepe::crepe! {
    @input
    struct Input(Fact);

    struct Fragment(Id);
    struct Edge(Id, Id, DispatcherType);
    struct ExternalReq(Id, DistId);
    struct SingletonReq(Id);
    struct Requirement(Id, DistId);

    @output
    struct Success(Id, Result);
    @output
    struct Failed(Id);

    // Extract facts.
    Fragment(id) <-
        Input(f), let Fact::Fragment(id) = f;
    Edge(from, to, dt) <-
        Input(f), let Fact::Edge { from, to, dt } = f;
    ExternalReq(id, dist) <-
        Input(f), let Fact::ExternalReq { id, dist } = f;
    SingletonReq(id) <-
        Input(f), let Fact::SingletonReq(id) = f;

    // Requirements from the facts.
    Requirement(x, d) <- ExternalReq(x, d);
    // Requirements of `NoShuffle` edges.
    Requirement(x, d) <- Edge(x, y, NoShuffle), Requirement(y, d);
    Requirement(y, d) <- Edge(x, y, NoShuffle), Requirement(x, d);

    // The downstream fragment of a `Simple` edge must be singleton.
    SingletonReq(y) <- Edge(_, y, Simple);

    // Multiple requirements conflict.
    Failed(x) <-
```

```
        Requirement(x, d1), Requirement(x, d2),
        (d1 != d2);
    // Singleton requirement conflicts with hash requirement.
    Failed(x) <-
        SingletonReq(x), Requirement(x, d),
        let DistId::Hash(_) = d;

    // Take the required distribution as the result.
    Success(x, Result::Required(d)) <-
        Fragment(x), Requirement(x, d), !Failed(x);
    // Take the default singleton distribution as the result,
    // if no other requirement.
    Success(x, Result::DefaultSingleton) <-
        Fragment(x), SingletonReq(x), !Requirement(x, _);
    // Take the default hash distribution as the result, if
    // no other requirement.
    Success(x, Result::DefaultHash) <-
        Fragment(x), !SingletonReq(x), !Requirement(x, _);
}
```

The system is also able to report results on success and has very little fixed overhead because as an embedded macro, it generates code that runs directly in the process, rather than requiring a separate OS process or language boundary.

## 6.3 DISTRIBUTED DATA PRIVACY MODEL CHECKER

The past examples have been in industry projects backed by companies aiming to sell a product. This final example is in Project Oak, an open-source research project on formal verification of security in distributed systems [11].

The `arcsjs-provable` package in Project Oak performs proofs on models from ArcsJs, a research secure application framework. It uses Crepe in the latest version at the time of writing, in March 2023. The primary usage is a logic program (in over 200 lines of code) that verifies a set of claims and capabilities over a graph. They use all of the features of Crepe, including custom data types from Rust, embedded expressions, and disaggregation.

Because the program is so long, it is not possible to summarize the usage in full in this paper. We refer the reader to the permanent link above if they are interested. However, we can still present some parts of it. All snippets below are inside the `crepe!` macro, and you can assume the `Ent` type represents a 64-bit integer ID.

```
struct CompatibleWith(pub Ent, pub Ent); // from, to

// Base case: just types.
CompatibleWith(x, y) <-
    KnownType(x),
    (!x.is_a(WITH_CAPABILITY)),
    KnownType(y),
    (!y.is_a(WITH_CAPABILITY)),
    Subtype(x, y);

// Check that y has the capabilities required by x.
CompatibleWith(x, y) <-
    KnownType(x),
    (x.is_a(WITH_CAPABILITY)),
    KnownType(y),
    HasCapability(y_cap, y), // For each capability y supports
    Subtype(x.args()[0], x_cap),
    Capability(x_cap, y_cap), // If supported we can continue.
    CompatibleWith(x.args()[1], y);

// If a type has no capabilities, discard the capabilities
// of its possible super type.
CompatibleWith(x, y) <-
    KnownType(x),
    (!x.is_a(WITH_CAPABILITY)),
    KnownType(y),
    (y.is_a(WITH_CAPABILITY)),
    CompatibleWith(x, y.args()[1]);
```

This set of three rules defines the `CompatibleWith` relation, which is part of a larger static analysis. It combines relations and facts from several data sources within a single grammar. Also, it's able to use user-defined methods on their custom data type like in the clause `x.is_a(WITH_CAPABILITY)`, which is a pure function that runs due to the embedding.

These kinds of complex rules combining multiple facets are common in static program analyses, such as the large analyses specified in the artifacts for Formulog [2]. Datalog allows the user to express this kind of computation succinctly using recursive queries.

Another interesting part of this logic program using Crepe is the `Subtype` relation, whose rules involve a lot of Rust embedding. Specific examples of this have been annotated as comments below.

```
Subtype(x, prod) <-
    KnownType(prod),
```

```
    (prod.is_a(PRODUCT)), // embedded conditional expression
    KnownType(x),
    // num_args() is a Rust function
    SubtypesAllArgs(x, prod, prod.num_args());

Subtype(prod, arg) <-
    KnownType(prod),
    (prod.is_a(PRODUCT)),
    for arg in prod.args(); // disaggregation

Subtype(labelled, labelled.args()[1]) <- // .args()[1] is Rust syntax
    KnownType(labelled),
    (labelled.is_a(LABELLED));
```

In another part of the subtyping rules of this program, we can see how including other Rust macros in expression position can be used to shorten some of the code.

```
Subtype(
    apply!(x_generic, x_arg),
    apply!(y_generic, y_arg)
) <-
    // ent! is a Rust macro, and it works fine
    Subtype(x_generic, ent!(GENERIC)),
    Subtype(x_generic, ent!(INDUCTIVE)),
    Subtype(y_generic, ent!(GENERIC)),
    Subtype(y_generic, ent!(INDUCTIVE)),
    Subtype(x_generic, y_generic),
    Subtype(x_arg, y_arg),
    KnownType(apply!(x_generic, x_arg)), // apply! is also a macro
    KnownType(apply!(y_generic, y_arg));
```

These examples indicate that the natural mechanisms for extending Crepe to interoperate better with and within Rust, its host language, are useful enough to appear in real-world code. Such features are only possible in embedded dialects of Datalog, and as Crepe is the first such embedded dialect that runs as a procedural macro, it aims to support them to their fullest extent by design.

# 7 EVALUATION OF CREPE

It's difficult to fully expound on the real-world characteristics of a programming language system for deductive inference, since by their expressive nature, logic programs tend to vary widely in the shape and structure of their computation. Each application of logic programming described in Chapter 4 has its own register of programming, with unique requirements for expressiveness and performance.

Crepe is a library that is meant to be embedded in Rust, and in that regard it has found its place in several applications as described in the last chapter. I will preface this chapter by saying that this is probably the best argument that can be given for showing that something as multi-dimensional as a programming language system is useful, as there are nebulous factors that go into any discussion of this. However, we can and should still do some basic performance and expressiveness benchmarks to evaluate the system and demonstrate that it works as intended.

## 7.1 PERFORMANCE BENCHMARK

Performance in Datalog engines has many factors. In the case of Crepe, the most comparable work include compiled Datalog engines like Souffle [10] and Formulog [2], as well as simple low-level embedded engines with restrictions, of which Datafrog is likely the best example [14].

Since Crepe doesn't have a single targeted use case, there is no representative benchmark that can be supported and run by all of the engines, so we'll instead share some basic results from a simple Datalog program run on inputs of different sizes. Consider the "graph walk" program below, which constructs a simple directed graph of edge density 0.04 and computes all of the paths in that graph of less than a certain length, as well as all pairs of nodes that do not have paths between them.

```
use crepe::crepe;

const MAX_PATH_LEN: u32 = 30;

crepe! {
    @input
    struct Edge(i32, i32, u32);
```

```
    @output
    struct Walk(i32, i32, u32);

    @output
    struct NoWalk(i32, i32);

    struct Node(i32);

    Node(x) <- Edge(x, _, _);
    Node(x) <- Edge(_, x, _);

    Walk(x, x, 0) <- Node(x);
    Walk(x, z, len1 + len2) <-
        Edge(x, y, len1),
        Walk(y, z, len2),
        (len1 + len2 <= MAX_PATH_LEN);

    NoWalk(x, y) <- Node(x), Node(y), !Walk(x, y, _);
}

fn walk(n: usize) -> (usize, usize) {
    let n = n as i32;
    let mut edges = Vec::new();
    for i in 0..n {
        for j in 0..n {
            if (i + j) % 50 < 2 {
                edges.push(Edge(i, j, 5));
            }
        }
    }

    let mut runtime = Crepe::new();
    runtime.extend(edges);
    let (walk, nowalk) = runtime
        .run_with_hasher::<fnv::FnvBuildHasher>();
    (walk.len(), nowalk.len())
}
```

Crepe is single-threaded because it is intended to be embedded in larger programs. I ran this program in Crepe on a computer with an ARM64 Apple M1 Pro processor and 16 GB of RAM. I also ported the Datalog parts of the same program to Souffle and Formulog, and I timed the resulting execution on a single thread in all three runtimes for $n = 32, 128, 512, 1024$

| ENGINE | CODE | COMPILE | NODES (n) | AVG. RUNTIME |
|---|---|---|---|---|
| **Souffle (I)** | 12 lines | — | 32 | 11.8 ms |
| *v2.4* | | | 128 | 32.3 ms |
| | | | 512 | 1.00 s |
| | | | 1024 | 7.54 s |
| **Souffle (C)** | 12 lines | 8.2 s | 32 | 10.0 ms |
| *v2.4* | | | 128 | 18.3 ms |
| | | | 512 | 284 ms |
| | | | 1024 | 1.99 s |
| **Formulog** | 12 lines | — | 32 | 1.89 s |
| *v0.7.0* | | | 128 | 2.68 s |
| | | | 512 | 7.38 s |
| | | | 1024 | 26.9 s |
| **Datafrog** | 36 lines | 0.46 s | 32 | 25.2 µs |
| *v2.0.1* | | | 128 | 1.78 ms |
| | | | 512 | 71.5 ms |
| | | | 1024 | 549 ms |
| **Crepe** | 12 lines | 0.69 s | 32 | 91 µs |
| *v0.1.7* | | | 128 | 3.3 ms |
| | | | 512 | 84.8 ms |
| | | | 1024 | 567 ms |

Table 7.1: Performance of Datalog engines on the graph walk benchmark.

nodes in the graph. Note that the number of edges in the graph is quadratic, asymptotically approaching $n^2/25$ where $n$ is the number of nodes. The results are shown in Table 7.1.

The "Code" column shows the total number of lines of code in the Datalog part of each implementation. This doesn't include code to generate the graph, massage the input into a form ("fact file") that the engine supports, or start the engine as a subprocess, since that obviously requires less code in Crepe because it is an embedded library. The "Compile" column shows how much time it took to compile the program, for engines with a separate compilation step. For all three of Souffle, Datafrog, and Crepe, we include linking of the final executable, but we don't include building dependencies because that is a one-time operation.

The engines labeled "Souffle (I)" and "Souffle (C)" are the interpreted and compiled versions of Souffle, respectively. Formulog has a compiled version, but that was not included due to instability in its current API. Datafrog was also compared, but because it is a low-level engine that requires the programmer to manually structure joins and optimize their computation, it required substantially more code. Compared to the other engines, which just asked for Horn clauses, the code for the Datafrog implementation was much more difficult to write. The Datapond tool can help scaffold Datafrog programs, but there's no way to fully generate the program from Horn clauses.

Nevertheless, Datafrog is valuable as a performance goal to aspire to, especially since it is also an engine that works well within larger Rust programs. It also uses an interesting optimized join algorithm.

Timings were measured by the sum of user and system time, meaning total time in computation. This is equal to wall clock time for most engines, since they are run in single-threaded mode. The only exception to this was Formulog, since although Formulog allows setting the parallelism option to 1, it still appears to spawn several threads. So for Formulog, the table includes the total execution time for all threads.

*These might be JVM threads, or threads used by Formulog for something else.*

Every benchamrk iteration was prefaced with `sync && sudo purge` to drop file system caches, for performance consistency. Every Datalog analysis had 2 warmup runs, followed by between 10-100 subsequent runs, and the results were averaged. The timings for interpreted engines (Souffle (I) and Formulog) include program parsing time, since parsing is required on each run if that variant of the engine is embedded within a larger software codebase.

## 7.2 DISCUSSION

Souffle and Formulog are both designed to run much larger analyses, and the design of their engines reflect this, with a lot of emphasis on features like concurrent data structures for efficient evaluation and an API that uses tab-separated "fact files" that are stored in a separate directory.

As a result, these engines are less suitable for the kinds of *embedded* deductive reasoning that Crepe is typically used for, and they have higher startup cost when evaluating the small to medium-sized inputs (graphs of up to 1024 nodes and 42000 facts) tested here. Souffle in particular is very slow to compile because it generates hundreds or thousands of lines of C++ code for even very short programs, which makes it harder to develop programs without relying on the interpreter. It's not uncommon to wait several minutes for Souffle to compile more complex programs.

From these examples, it appears that Crepe outperforms the two program analysis-focused engines, Souffle and Formulog, at this embed-

ded logic programming task. Crepe also has comparable performance to Datafrog, although it is not quite able to match Datafrog's speed, especially for small inputs. However, Datafrog is a very low-level engine that only provides bits and pieces, while the burden of optimization and actually implementing nontrivial queries is put on the user of the library. Even so, despite the large differences in ease-of-use, Crepe is only around 10-20% slower than Datafrog on this benchmark for $n = 512, 1024$ and between 2-4x slower for $n = 32, 128$.

*Note that Crepe and Datafrog use different algorithms, so it's an interesting coincidence that their total runtimes are similar on the larger inputs.*

From this benchmark, running engines like Souffle or Formulog within a larger application seems difficult. Besides the annoyance of having to create your own fact files within a directory and launch the subprocess, there is also significant overhead to all of these extra steps, as shown in the runtime being over 100-10000 times slower for the benchmarks on a graph with 32 nodes. If a larger application were running such a program on many small inputs, as in the case studies from Section 6.1 and Section 6.2, this fixed-cost overhead would likely be unacceptable.

Thus, Crepe fills a void in the existing space of Datalog dialects by being simple and direct to use, unlike Datafrog, while also remaining very fast, pragmatic, and easy to embed in larger programs, unlike standalone Datalog engines for program analysis. It's able to achieve this unique combination of simplicity and expressiveness through careful language design around seamless compilation of procedural macros.

# Part III

# LANGUAGES FOR EXPLORATORY DATA ANALYSIS

# 8

# REACTIVE NOTEBOOKS FOR DATA ANALYSIS AND VISUALIZATION

One of the most substantial questions in modern science is how to make data analyses *accessible* to a broader audience. Accessibility is a serious concern for data systems because conclusions drawn from data often have far-reaching consequences, so it is important for scientists to not just communicate the conclusions they find from a data study, but also to give others the ability to verify, reproduce, and extend their results.

Interactive notebooks are a common way of sharing literate programs with documentation, graphics, and computations. In data science, one of the most popular tools in this category is Jupyter [17], a software platform that aims to promote "open science" by allowing programmers to develop multimedia documents where the underlying source code (Python or R) is completely visible. However, Jupyter notebooks have design choices that limit reproducibility, since they necessarily rely on highly technical aspects of the user's computer: installed software, hardware type, drivers (e.g., GPU availability), operating system, programming language version, file system, access to data, networking, and numerous other concerns.

Additionally, Jupyter notebooks typically run code in a supported kernel language, such as Python or R. However, these languages are imperative by nature and do not natively support declarative transformations for relational data. Even with libraries provide native domain-specific query languages for interacting with data, such as Pandas [12] and Dplyr [30], these APIs are fundamentally constrained by the limits of their host language and therefore cannot provide a fully declarative interface, like SQL databases.

To resolve these issues, in this part, I propose that exploratory data analysis should ideally be executed in an environment that is:

- **Declarative:** Allows users to analyze, clean, manipulate, and visualize data within a single context, with minimal technical "glue" overhead.
- **Interactive:** Is simple, tangible, and easy to play with, since data analysis is a process that is fundamentally based on exploration.
- **Reproducible:** Has minimal overhead for software installation, uses permanent, public addresses for data, and can be run and modified by anyone with access to the notebook.

To meet these goals, we introduce Percival, a language for declarative data analysis and visualization. Percival builds on Datalog as a core, extending it with reactive features, aggregations, and embedded JavaScript. Percival also integrates with modern, declarative plotting libraries (based on D3 [3]) and seamlessly propagates reactive execution through these components.

One inconvenience with past Datalog implementations is that they have been tied to complex domain-specific databases (see Section 4.4) or runtime environments (see Section 4.1). In contrast, Percival is able to import data from standard web formats (JSON, CSV) and uses a novel method of compiling queries just-in-time to JavaScript, enabling a more accessible editing experience. As a consequence, Percival is also the first compiled Datalog engine that runs entirely in standard web browsers, without requiring communication with a background server.

## 8.1 DESIGN GOALS

Any system for data analysis needs to allow users to load data from one or more sources, calculate with and transform that data, generate visuals, and share the resulting analysis with others. Most data analyses are somewhat complex, so the user needs to write code in order to deal with different formats and flexibly express other common operations, such as statistical algorithms and parsing. When code is involved, libraries are often designed to help programmers quickly write efficient code.

The focus of this project was to explore connections at the language level between Datalog and data analysis. However, such changes need to be evaluated holistically due to how complex the data analysis process is.

Therefore, we chose a literate programming interface that combines code with text as the primary method of interacting with our language. The user interface contains no hidden state, and it is presented as an interactive notebook, with three types of cells:

- **Code cells:** Contain Percival programs in Datalog syntax, which is where queries are written, outputting the result of queries.
- **Markdown cells:** Contain documentation and rich text description, displaying a rendered view of the Markdown that automatically updates as the user types.
- **Plot cells:** Contain code for data visualization, outputting a rendered chart in interactive SVG format.

Percival is a client-side web application running fully in the user's browser. The notebook frontend is built with general-purpose web frameworks (Svelte, Tailwind CSS) and relies on other open source libraries, including

CodeMirror 6 for live code editing and syntax highlighting, Remark for Markdown rendering, and KaTeX for LaTeX math support.

This information is provided for context. An interested reader can see Percival's source code for more details on the software engineering and interface design components, but those are not the focus of this thesis. In this section we discuss the design goals for the Percival language itself, i.e., the just-in-time compiled dialect of Datalog used in code cells.

### 8.1.1 *Exploration versus engineering*

Percival's design fundamentally differs from Crepe in its primary purpose: while Crepe is meant for careful embedding within a system, Percival is designed for exploratory tasks. These differing objectives highlight the duality of computer usage, serving both as tools for building applications and as platforms for understanding digital information.

For example, engineering systems demand reliability, testability, and extensibility, whereas data analysis prioritizes expressiveness, flow state, and ease of presenting information. In systems programming, writing one or even five extra lines of code to use a library may not be a significant concern, but in data analysis, this can become a substantial annoyance, impeding the flow of exploration. Therefore, Percival accents how Datalog can flexibly and naturally express relational programming on data.

Visualization plays a critical role in data analysis tools, as it leverages our eyes' high-throughput sensory capabilities. By representing data in a two-dimensional space, users can quickly grasp patterns, trends, and outliers that might otherwise be difficult to discern.

Data exploration often involves rapid iteration and experimentation, requiring tools that can accommodate quick changes and updates. Code in this context will be written once but edited multiple times interactively, emphasizing the need for an environment that supports rapid modification and real-time feedback.

Tight feedback loops are also important in other usage contexts. Interactive visualizations, similar to those published by the New York Times, exemplify the appeal of engaging with data directly. By allowing users to touch and manipulate data, they foster a deeper understanding of the information being presented.

### 8.1.2 *Experiments around reactivity*

Another design goal of Percival is to see how reactive computing workflows fit in with Datalog as a language. Reactivity has a rich history in the programming languages literature, and its effects can be seen in many

of the modern UI frameworks and building blocks that programming commonly use today.

When reactivity is presented here though, it refers to a particular application, more concrete than the state-of-the-art of theory. We consider how reactivity can help programmers in the context of interactive data notebooks, where people work quickly and often need to change code to adjust their analysis as they make new observations.

Imagine a spreadsheet software application. The data is laid out fully in front of the user in cells, and they are allowed to perform computations on that data using formulas. Formulas can depend on each other, just like in programming languages. The crucial feature of spreadsheets is that if you change a cell, all cells that depend on it will also be recomputed automatically, as well as cells depending on those, and so on.

Newer notebook software like Observable, Pluto.jl, and Hex adopt this model to avoid the limitations of classic imperative-style notebooks like Jupyter. In Jupyter notebooks, the order in which cells are executed affects the semantics, and users must manually rerun all cells if the first one changes. This can be useful in situations where code is slow to run, such as machine learning, but it creates unnecessary friction for exploratory analysis of small datasets. Moreover, if a cell that added or mutated variables is deleted, those changes persist even after the cell's removal.

Reactivity allows users to see all the state in the notebook simultaneously, similar to reactive libraries' appeal for frontend development. However, adding reactivity to most languages is not trivial. For example, Python's dynamic nature resists static analysis to track data dependencies, and Jupyter has a vast ecosystem of third-party extensions that would likely be incompatible. Notebooks like Observable implement a source code transpiler on top of JavaScript and provide rules to the programmer. Similarly, Pluto.jl restricts its code cells to define exactly one variable and uses Julia's built-in code parser to then analyze dependencies.

To incorporate reactivity as a feature, Percival must add it to Datalog. Fortunately, the language is simple enough to enable this. Each cell is implemented as a separate *stratum* of execution, which has tradeoffs but is flexible enough for most analyses that are typically step-by-step. Reactivity and stratification also allow Percival to isolate each execution in a separate Web Worker thread and run them concurrently, enabling termination of threads that are stale, crash with errors, or enter an infinite loop.

## 8.2 THE PERCIVAL LANGUAGE

The syntax of Percival is compactly and unambiguously summarized by the extended Backus-Naur form grammar in Fig. 8.1. This section will

```
(* Note: jsexpr_, number_, string_, identifier_, and boolean_ are
   tokens from lexical analysis. *)

program = { statement };
statement = rule | import;

rule = literal, "." | literal, ":-", clauses, ".";
clauses = { clause, "," }, clause;
clause = literal | jsexpr_ | binding;
literal = identifier_, "(", [ { prop, "," }, prop ], ")";
binding = identifier_, "=", value;
prop = identifier_, [ ":", value ];

value = identifier_ | primitive | jsexpr_ | aggregate;
aggregate = identifier_, "[", value, "]", "{", clauses, "}";
primitive = number_ | string_ | boolean_;

import = "import", identifier_, "from", string_;
```

Figure 8.1: Simplified EBNF grammar for the Percival language.

provide guided examples of the syntax, while commenting on syntactic and semantic differences from other existing Datalog implementations.

### 8.2.1 *Basic syntax*

At its core, a Percival program (as a dialect of Datalog) is a series of rules written as Horn clauses. In this basic capacity, Percival draws most of its core semantics directly from Datalog. The simplest example is a program that computes the transitive closure of a directed graph.

```
edge(from: "foo", to: "bar").
edge(from: "bar", to: "baz").

path(from, to) :- edge(from, to).
path(from, to) :- edge(from, to: z), edge(from: z, to).
```

The expected behavior of this program would be that it derives three facts, written below.

```
path(from: "foo", to: "bar").
path(from: "bar", to: "baz").
path(from: "foo", to: "baz").
```

Here, Horn clauses are used to construct relational queries. Notice however that unlike Prolog, Percival uses named fields to represent relations. This

has advantages for representing the types of data common in data analysis tasks, which is often in the structure of named records and tables, but the difference in syntax may appear jarring at first. When a field name is referred to without a value, as in `tc(from, to)`, it is a syntactic shorthand for the literal `tc(from: from, to: to)`.

### 8.2.2 *Embedding JavaScript*

Although Horn clauses are already enough to arrive at a pure Datalog language with well-behaved semantics and guaranteed termination, practical data analyses require a richer class of primitives to allow programmers to perform computation. Rather than constructing a new domain-specific language for this, Percival allows users to embed JavaScript at necessary locations in their code, between backquote `` `...` `` characters. For example, the program below uses the built-in `Math.sqrt()` function and string concatenation.

```
name(full_name: `first + " " + last`, sqrt_age) :-
  person(first, last, age),
  sqrt_age = `Math.sqrt(age)`.

person(first: "Alice", last: "Carol", age: 20).
person(first: "Foo", last: "Bar", age: 45).
person(first: "Baz", last: "Lam", age: 12).
```

Because Percival runs within the user's browser (more on this in Section 8.3), user-provided JavaScript can be embedded with the Percival compiler's output, similar to how a C compiler allows programmers to write inline assembly syntax. This approach was inspired by previous work on embedding Rust in Crepe, described in Part II. The resulting JavaScript code is executed by the browser's JIT engine in tandem embedded in Datalog inference logic.

Some clauses in a rule are temporary *bindings* such as `sqrt_age` in the example above, which define a variable in the context of the current rule's execution. This was done for efficiency, to reuse computation in the case of repeated expressions, and to allow programmers the flexibility to rename their variables. It also means that Percival data types are identical to the data types of the environment's underlying JavaScript engine, which reduces interpretation overhead.

If an embedded JavaScript expression is placed in clause position without being in a binding, it is treated as a filtering operation for that rule. For example, the following program computes all walks of length at most 10 in a directed graph.

```
walk(from: v, to: v, len: 0) :- edge(from: v).
walk(from: v, to: v, len: 0) :- edge(to: v).

walk(from, to, len) :-
  walk(from, to: z, len: len1),
  edge(from: z, to),
  len = `len1 + 1`,
  `len <= 10`.
```

To show one more example of the utility of embedding expressions from the host language, here is a simple program in Percival that computes the values of the first 30 Fibonacci numbers.

```
max_n(value: 30).

fib(n: 0, v: 0).
fib(n: 1, v: 1).
fib(n: `n + 1`, v) :-
  fib(n, v: v1),
  fib(n: `n - 1`, v: v2),
  v = `v1 + v2`,
  max_n(value),
  `n < value`.
```

Notice above that the `max_n` relation is used as a constant. By changing the value provided in the relation, the programmer can interactively modify the limits of the index `n` that the computation goes up to, while remaining within the logic programming framework.

### 8.2.3 *Data imports*

Percival allows the user to load any public JSON, CSV, or TSV dataset from GitHub, NPM, or HTTPS web link, although the latter option is subject to the same-origin policy. In the following example, we import a publicly available dataset on cars from NPM.

*This internally uses jsDelivr, a free content delivery network for open source files.*

```
import cars from "npm://vega-datasets@2.1.0/data/cars.json"
```

Note that `npm://` and `gh://` package imports have an optional tag that can be used to specify the specific version, release, or Git commit hash that the data should be loaded from. Given a JSON file, the contents should be an array of objects that are loaded into a relation by the given name. CSV and TSV files are handled similarly, although the data types in the file are also inferred to make loading numerical data easier.

### 8.2.4 *Aggregation*

Many practical data analyses will often include one or more levels of *aggregation*, to combine multiple data rows into a single output by applying a mathematical operator. Percival supports aggregates through a nested subquery syntax. Currently, the supported aggregates are `sum`, `min`, `max`, `mean`, and `count`. For example, the following query finds the average mileage of all cars in the dataset, partitioned by year and country of origin.

```
average_mpg(country, year: `new Date(year)`, value) :-
  country(name: country),
  cars(Year: year),
  value = mean[Miles_per_Gallon] {
    cars(Origin: country, Year: year, Miles_per_Gallon)
  }.
```

It is important to note that aggregates are placed in value position, and the subqueries in aggregate syntax run a fully enabled set of features in the Percival syntax. However, unlike top-level queries, relations mentioned in subqueries are required to belong to the dependencies of a program cell; they cannot belong to the intensional database. This is a manual realization of *stratified* code to avoid compiler errors (see Section 2.4). The Percival prototype does not yet compute strongly connected components, requiring the user to split their strata between multiple cells instead, although this is not a fundamental limitation.

Also, aggregates in Percival are a superset of stratified negation. Negations of literals on the right-hand side of a rule can be simulated using aggregates by simply checking that the `count` value of a trivial subquery equals zero, using an embedded JavaScript equality comparison.

*Souffle and some other systems only support one level of aggregation.*

Since aggregation is so particularly common in data analysis, Percival also supports nested aggregates or subqueries. The following program, using the `student(name)` and `assignment(author, grade)` relations, computes the total sum of the average grade each student received on their assignments.

```
result(value) :-
  value = sum[highest_grade] {
    student(name),
    highest_grade = mean[grade] {
      assignment(author: name, grade)
    }
  }.
```

Finally, it is worth noting that Percival can be very easily extended with new aggregates, since each operation is implemented as a simple

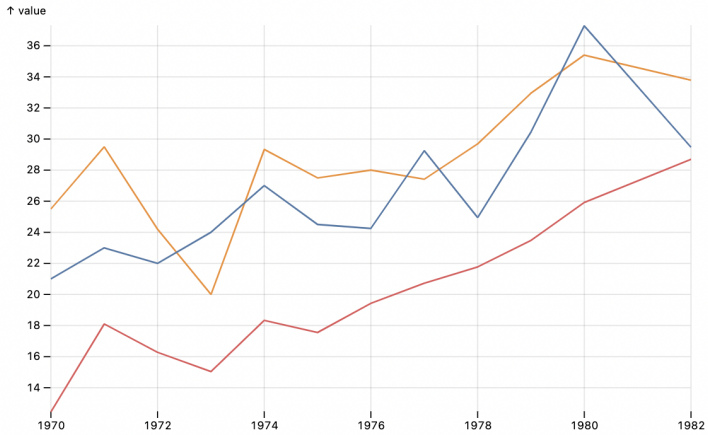
function in JavaScript that is passed to the runtime. For example, the current implementation of the five basic aggregate operations in TypeScript is shown below. A future version of Percival could also allow users to define their own custom aggregate functions by embedding this kind of imperative specification.

```
const aggregates: Record<string, (results: any[]) => any> = {
  count(results) {
    return results.length;
  },
  sum(results) {
    return results.reduce((x, y) => x + y, 0);
  },
  mean(results) {
    return results.reduce((x, y) => x + y, 0) / results.length;
  },
  min(results) {
    let min = null;
    for (const x of results) {
      if (min === null || x < min) {
        min = x;
      }
    }
    return min;
  },
  max(results) {
    let max = null;
    for (const x of results) {
      if (max === null || x > max) {
        max = x;
      }
    }
    return max;
  },
};
```

## 8.3 SANDBOXED WEB RUNTIME

Percival programs are written in an interactive notebook interface that runs in any web browser. The declarative specifications are just-in-time translated to imperative JavaScript code and executed within a worker thread of that same web browser, never leaving the user's computer.

Percival uses automatic index generation, a semi-naive evaluation strategy, and immutable set data structures to run Datalog programs quickly, similar to the staged compilation approach taken by Crepe. The Percival compiler (i.e., the Datalog-to-Javascript translator) is written in Rust, and it can be compiled to WebAssembly to run in the browser itself.




```
average_mpg => Plot.line(average_mpg, {
  sort: "year",
  x: "year",
  y: "value",
  stroke: "country",
}).plot({ grid: true })
```

Figure 8.2: Simple time-series line chart drawn in a Plot cell with Percival.

We will omit sharing an example of the generated JavaScript code for brevity, as it is quite complicated, but the results of the translation are very similar to the program that was walked through in Section 5.3. Declarative logic specifications are translated into JavaScript code, which pulls in dependencies and language intrinsics, including the Immutable.js library, and then the semi-naive evluation code is executed.

Another aspect of the web notebook environment is data visualization. Visualization is a key tool for exploring complex datasets and effectively communicating results to others. As mentioned previously, Percival allows the user to include *Plot* cells, which run JavaScript code that generates diagrams using the Observable Plot library. This could be easily extended to support any declarative plotting library on the web.

A simple example of a Plot cell is shown in Fig. 8.2. The syntax of the cell consists of a relation name, followed by the => token, and then a JavaScript expression that returns an SVG vector plot that can be rendered by the browser. Using such a method to integrate external libraries, Percival is able to rely on modern data visualization tools to parse declarative specifications, which enables a great deal of freedom in their expressive qualities.

We will now discuss how and in what order code is executed. First, to make Percival's runtime reactive, code in a notebook is split between multiple cells, and when one cell uses a relation defined in another, the

Percival runtime will construct a dependency graph and execute the programs in reverse topological order based on which dependencies have changed. This is analogous to stratification discussed in Section 2.3.

As an example of this behavior, if a user had two cells where one depended on the other, such as one cell that imports the `cars` dataset, and another cell that runs an aggregate query on it, then changing either cell would rerun both cells in order, while changing the aggregate cell would only rerun that cell individually. We choose to enforce that cells respect stratification, in that two cells cannot belong to the same stratum or contain any rules that generate the same relation, to make cross-cell dependencies more clear for the programmer.

Because Percival runs entirely within a browser environment, security and performance is also a concern. If a long-running computation such as an infinite loop were to occur in a notebook, it would be unacceptable for this computation to block the browser's main thread, as this would cause the user interface to freeze and the website to eventually crash. Furthermore, Datalog execution is inherently parallelizable between different cells, and it should be possible to cancel and restart a computation if some of its dependencies change in the middle of execution.

To address these issues, Percival relies on Web Workers, a standard browser technology, to run compiled Datalog queries in a sandboxed environment on separate operating system threads. Input data from dependent relations is sent to the worker through a channel, and the results from evaluating the compiled program are sent back after execution is finished. Furthermore, errors are propagated back up to the user interface in real time, and worker threads are terminated on demand from the main thread in case of stale dependencies.

It should also be mentioned that Plot cells receive the same reactive treatment as Datalog evaluation, and they are also included in the dependency graph as sink nodes. This is what allows graphs to update in real time as their source relations are changed. It is particularly important to run Plot cell code in web workers because they directly include user-provided JavaScript functions, rather than single expressions. To allow libraries like D3 to run in a browser context, the plot worker patches its global document object model (DOM) with a lightweight virtual implementation.

*Specifically, Percival uses the Domino library for a virtual DOM.*

## 8.4 DATA ANALYSIS DEMO

The easiest way to experience Percival is by example. An interactive demo showing how all of these pieces fit together to work on a real-world dataset is available online at the project website, `https://percival.ink/`. This notebook runs completely in the browser, deterministically regenerating its outputs from only the saved source code in under a second. For the

sake of being self-contained, we reproduce the main demo in the following two sections of this text. The outputs of each cell are shown above the code, as in the web interface.

### 8.4.1 *Exploring airport data*

All data analysis starts with data. For the sake of this demonstration, consider the `airports.csv` file from vega-datasets, an open-source repository of example datasets. After loading this file into Percival, it forms an `airports` relation with the following structure.

```
Table{airports} := [
  airports(iata: "AIG", name: "Langlade County", city: "Antigo", state:
    "WI", country: "USA", latitude: 45.15419444, longitude:
    -89.11072222),
  airports(iata: "DEQ", name: "J Lynn Helms Sevier County", city: "De
    Queen", state: "AR", country: "USA", latitude: 34.04699556,
    longitude: -94.39936556),
  airports(iata: "X16", name: "Vandenberg", city: "Tampa", state: "FL",
    country: "USA", latitude: 28.01398389, longitude: -82.34527917),
  airports(iata: "MOB", name: "Mobile Regional", city: "Mobile", state:
    "AL", country: "USA", latitude: 30.69141667, longitude:
    -88.24283333),
  airports(iata: "4O4", name: "McCurtain County Regional", city:
    "Idabel", state: "OK", country: "USA", latitude: 33.909325,
    longitude: -94.85835278),
  3371 more items...
]
```

```
1 import airports from "npm://vega-datasets@2.1.0/data/airports.csv"
```

Notice that this dataset contains several thousand rows, which would be difficult to ergonomically load and manipulate within a typical graphical spreadsheet application. It also has several types of data, in both numerical and textual modes. An initial question to ask is: *How many airports are there within each country described by this dataset?* This can be answered by a simple query.

```
Table{airports_per_country} := [
  airports_per_country(count: 3372, country: "USA"),
  airports_per_country(count: 1, country: "Palau"),
  airports_per_country(count: 1, country: "N Mariana Islands"),
  airports_per_country(count: 1, country: "Thailand"),
  airports_per_country(count: 1, country: "Federated States of
    Micronesia"),
]
```

```
1 airports_per_country(country, count) :-
2   airports(country),
3   count = count[1] { airports(country) }.
```

There are some syntactic points above. The `1` in `count[1]` is an arbitrary expression and could be replaced with just `count[]` in the future. Also, the `airports(country)` literal is really shorthand for something like `airports(country, iata: _, name: _, city: _, ...)`, where all of the other named fields are ignored. This allows the rule to bind to all unique countries in the dataset.

With this aggregate query, Percival is able to compute the count statistics of airports by country and stores that within a new table. There is an automatic dependency as well, so whenever `airports` changes, this cell will be invalidated and rerun to compute the new value of `airports_per_country` as well. Looking at the outputs, it seems like all but 4 of the airports are in the United States. The next step would be to filter airports within the United States and reduce to only 3 columns that are relevant for analysis at the moment.

```
Table{us_airports} := [
  us_airports(iata: "FYE", name: "Fayette County", state: "TN"),
  us_airports(iata: "SXQ", name: "Soldotna", state: "AK"),
  us_airports(iata: "S21", name: "Sunriver", state: "OR"),
  us_airports(iata: "LKV", name: "Lake County", state: "OR"),
  us_airports(iata: "MRB", name: "Eastern Wv Reg/Shephard", state: "WV"),
  3367 more items...
]
```

```
1 us_airports(state, iata, name) :-
2   airports(state, iata, name, country: "USA").
```

After this query, the cleaner `us_airports` relation can be queried to find aggregate statistics for number of airports by state. It's also easy to make a simple, reactive line plot visualizing the results of the query.

```
Table{airports_per_state} := [
  airports_per_state(count: 205, state: "CA"),
  airports_per_state(count: 88, state: "IL"),
  airports_per_state(count: 11, state: "PR"),
  airports_per_state(count: 1, state: "GU"),
  airports_per_state(count: 102, state: "OK"),
  52 more items...
]
```

```
airports_per_state(state, count) :-
  us_airports(state),
  count = count[1] { us_airports(state) }.
```


```
airports_per_state => Plot.plot({
  marks: [
    Plot.dot(airports_per_state, {
      x: "count",
      fill: "steelblue",
      fillOpacity: 0.6,
    }),
  ],
  grid: true,
})
```

In this exploratory visualization, we can make some immediate conclusions about the data. It seems like most states have between 0-100 airports, with a few outliers having 200-300 airports. This makes sense, given that some states are much smaller than others, and even between states of the same size, population density can be very different! Notice how to arrive at this conclusion, the notebook combines the use of Datalog as a declarative data query tool (to perform transformations: aggregates and filtering) and an integrated third-party declarative plotting library.

### 8.4.2 *Joins with census data*

Although we could continue to explore the current dataset and gain more understanding for how the data is structured, at some point it is natural to ask questions that require other sources of data as well. For example, how does the number of airports compare to the land area of the state or the human population? This is a common instance of a pattern in declarative data analysis where *multiple* sources are combined, often using different transformations and origins, to arrive at a desired result.

Percival supports pulling data from multiple sources transparently within its Datalog query syntax. It also can track reactive dependencies through its execution model. To demonstrate this, we first load state abbreviations, areas, and population data from three external sources with minimal prior data cleaning.

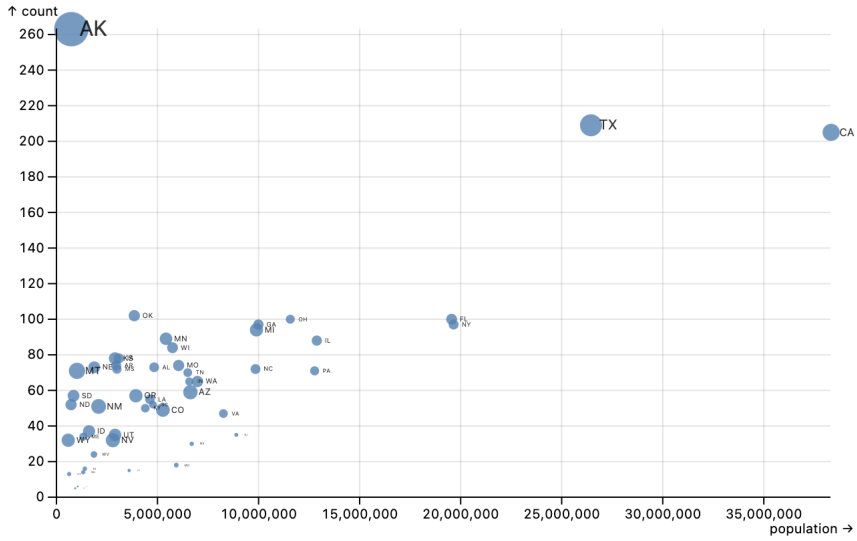
```
Table{state_abbrevs} := [
  state_abbrevs(state: "Alaska", abbreviation: "AK"),
  state_abbrevs(state: "Pennsylvania", abbreviation: "PA"),
  state_abbrevs(state: "Virginia", abbreviation: "VA"),
  state_abbrevs(state: "Maine", abbreviation: "ME"),
  state_abbrevs(state: "Montana", abbreviation: "MT"),
  46 more items...
]

Table{state_areas} := [
  state_areas(state: "Virginia", area_sq_mi: 42769),
  state_areas(state: "Delaware", area_sq_mi: 1954),
  state_areas(state: "Arizona", area_sq_mi: 114006),
  state_areas(state: "Michigan", area_sq_mi: 96810),
  state_areas(state: "Missouri", area_sq_mi: 69709),
  47 more items...
]

Table{state_population} := [
  state_population(state: "NH", ages: "total", year: 1990, population:
    1112384),
  state_population(state: "MI", ages: "total", year: 2007, population:
    10001284),
  state_population(state: "SC", ages: "under18", year: 2009, population:
    1079729),
  state_population(state: "CA", ages: "under18", year: 2010, population:
    9284094),
  state_population(state: "TX", ages: "under18", year: 2002, population:
    6060372),
  2539 more items...
]
```

This data can be loaded easily but doesn't start out in a clean format. For example, the `state_areas` table has full state names, while the `state_population` table is grouped by two-letter abbreviations. Also, the `state_population` table has data fields for multiple years and age groups, which might be more than we might need for a basic analysis. The programmer must filter and transform this data to effectively use it. Next, we compute the population, land area, and number of airports in each state.

```
Table{airports_state_info} := [
  airports_state_info(area: 656425, count: 263, population: 735132,
    state: "AK"),
  airports_state_info(area: 1954, count: 5, population: 925749, state:
    "DE"),
  airports_state_info(area: 84904, count: 35, population: 2900872, state:
    "UT"),
  airports_state_info(area: 9615, count: 13, population: 626630, state:
    "VT"),
  airports_state_info(area: 42769, count: 47, population: 8260405, state:
    "VA"),
  46 more items...
]
```

```
1 airports_state_info(state, count, population, area) :-
2   state_abbrevs(state: name, abbreviation: state),
3   airports_per_state(count, state),
4   state_population(state, population, ages: "total", year: 2013),
5   state_areas(state: name, area_sq_mi: area).
```

Unlike most imperative data cleaning workflows, this query was not destructive in any way, as none of the original datasets were mutated. It's also very easy to take a specific slice of the population dataset by simply specifying the values `ages: "total"` and `year: 2013` to filter the table by two columns. The block finishes with a simple Datalog clause to match state names with their abbreviations, which would otherwise be more complex in imperative code.




```
airports_state_info => Plot.plot({
  marks: [
    Plot.dot(airports_state_info, {
      x: "population",
      y: "count",
      r: "area",
      fill: "steelblue",
      fillOpacity: 0.8,
      title: "state",
    }),
    Plot.text(airports_state_info, {
      x: "population",
      y: "count",
      textAnchor: "start",
      dx: 6,
      text: "state",
      fillColor: "#222",
      fillOpacity: 0.8,
      fontSize: d => Math.sqrt(d.area) / 50,
    }),
    Plot.ruleY([0]),
    Plot.ruleX([0]),
  ],
  grid: true,
})
```

The final cell of this example visualizes the results of the aggregate query within a scatter plot. The labeled column-based tabular relation syntax lends itself well to modern declarative plotting grammars, as shown above, where the four fields named `population`, `count`, `area`, and `state` are each manifested through different visual components such as color, location, size, and text. This makes it possible to express complex plots using the same reactive interface.

# 9 EVALUATION OF PERCIVAL

In this section, we evaluate Percival as a language. Our prototype aims to verify two principal aspects: first, that the language is reasonably fast enough for data analyses, although not necessarily the fastest method, and second, that it is expressive enough to comfortably manipulate data and answer common exploratory questions.

## 9.1 PERFORMANCE BENCHMARK

To check that Percival executes reasonably quickly, we'll run a performance microbenchmark, using the same "graph walk" program described in Section 7.1.

Note that walking graph data structures is not a common data analysis program, and furthermore, Percival is based on web technologies and compiles to JavaScript rather than directly to optimized Rust or C++ code. It also uses less efficient immutable data structures while in this current prototype. However, the comparison still should be done, as it verifies that the speed is reasonable and asymptotically consistent. The results are shown in Table 9.1.

As seen in the table, the Percival prototype is significantly slower than interpreted Souffle, since it runs within web browsers and makes liberal use of JavaScript. However, the slowdown is within only approximately an order of magnitude of difference, and that only shows up for larger inputs involving around a billion computations (since the graph walk program is $O(n^3)$), which is more than in most data analyses that the prototype is targeted towards.

This indicates that Percival in its current prototype stage is not well-optimized yet, and further work could be done to make it faster. However,

| NODES ($n$) | SOUFFLE (I) | PERCIVAL | SLOWDOWN |
|---|---|---|---|
| 32 | 11.8 ms | 9.67 ms | 0.8× |
| 128 | 32.3 ms | 89.6 ms | 2.8× |
| 512 | 1.00 s | 7.96 s | 8.0× |
| 1024 | 7.54 s | 89.8 s | 12× |

Table 9.1: Relative speed of Percival on the graph walk benchmark.

for the intended use case of interactive data visualizations, computations that take less than 100 ms have an effectively imperceptible update delay, and this has been the case for the data analyses in our example notebooks for Percival so far. We omit further performance benchmarks to leave room for WebAssembly-based explorations in the future, since the current prototype's speed is reasonable but not competitive.

## 9.2 EXPRESSIVENESS BENCHMARK

Next, we will see how the Percival language is able to handle common tasks from the expressiveness benchmark of [5]. This is a benchmark of tabular data analysis problems solved in various programming languages and domain-specific language systems for the purposes of comparison. It's useful to see how these problems would be solved in Percival.

The benchmark has 15 problems divided into 6 categories: aggregation, joins, strings, first-order logic, time series, and graphs. For the sake of brevity, we will solve the first problem in each category. The first problem is to find the continent with the highest average population by country. This can be solved using two cells in Percival.

```
average_population(continent, value) :-
  countries(continent),
  value = mean[population] { countries(continent, population) }.

highest_population(continent) :-
  max_population = max[value] { average_population(value) },
  average_population(continent, value),
  `value === max_population`.
```

This query shows how you can compose aggregates together to form a more complex query. However, it would have been simpler to express this query if Percival had support for SQL-like `LIMIT` and `ORDER BY` clauses, as then the second aggregate would not have been necessary.

The second problem is, given a table of actors and of directors, to find all the directors of movies that Tom Hanks starred in. The solution is simple to express in Datalog.

```
tom_hanks(director) :-
  actors(actor: "Tom Hanks", movie),
  directors(director, movie).
```

The third problem is to convert the string value in each row of a table to a number, removing commas if the format column is a specific string, and otherwise removing underscores. To express this in Percival, it's easiest to

just rely on an embedded Javascript snippet, since conversions like this are exactly what general-purpose scripting languages designed for.

```
strings_to_numbers(n) :-
  numbers(format, value),
  n = `Number(value.replace(
    format === "under_sep" ? /_/g : /,/g, ""))`.
```

The fourth problem is to find all buyers of food that ordered every food item at least once, listed in a separate table by ID. This rqeuires two cells (strata) and makes use of counts to check that the buyer ordered every food item.

```
orders_unique(buyer, food) :- orders(buyer, food).

all_purchased(buyer) :-
  orders_unique(buyer),
  num_purchased = count[1] { orders_unique(buyer) },
  num_total = count[1] { food() },
  `num_purchased === num_total`.
```

The first rule in this program may seem confusing at first, but it is removing the `id` column from the orders table to only filter by unique orders from each buyer for a given food item. This would not be necessary if there was a `count_unique` aggregate in the language.

The fifth problem asks to compute 7-day rolling averages within a time series of data points. This is quite concise in Percival.

```
rolling(end_time, average) :-
  data(time: end_time),
  average = mean[x] {
    data(time, x),
    `end_time - 7 < time && time <= end_time`
  }.
```

Finally, the sixth problem is transitive closure in a directed graph, returning all vertices reachable from a given vertex.

```
reachable(node) :- query(source: node).
reachable(node) :-
  reachable(node: prev),
  graph(source: prev, target: node).
```

## 9.3 DISCUSSION

Percival is currently a prototype and offers significant room for performance optimization. For instance, the slow immutable data structure library could be implemented in WebAssembly instead of JavaScript, which

would greatly reduce overhead during semi-naive evaluation loops. Furthermore, the entire Percival compiler could target WebAssembly rather than JavaScript, potentially bringing the system closer to native Datalog execution speeds.

The expressiveness benchmark demonstrated that Percival is effective at solving various non-cherry picked problems, which was the original goal. Logic programming is not usually applied to such tabular data analysis tasks and certainly not in an interactive notebook environment. Percival shows that it is possible to fit Datalog as the core of a query language for data exploration tasks.

However, the expressiveness benchmark also demonstrated that Percvial still felt awkward in some areas. Continued iteration would further refine the language. For example, the syntax for aggregation and labeled entries in tables is verbose, and the JavaScript embedding syntax could be better integrated into the language, since it proves quite useful as an expression language within Percival for many of the evaluated tasks.

Additionally, requiring every cell to be an individual stratum is overly restrictive. Percival should perform stratum splitting using strongly connected components within each cell, resulting in a faster runtime that does not force users to unnecessarily split aggregates between cells. This is a technical limitation of the prototype that can be solved with more implementation work. Several other core features are useful in a data analysis system like limits, result sorting, and user-defined aggregates, which are currently not implemented in Percival.

# Part IV

# CONCLUSION

# 10 CONCLUSION

Throughout this thesis, we have explored the unique characteristics of Datalog as a programming language, emphasizing its simplicity, composability, and power. We designed and implemented two systems based on Datalog, each addressing different domains of computer science, and evaluated their benefits and drawbacks.

In Part II, we presented Crepe, a high-performance Datalog implementation embedded within the Rust programming language. By deeply integrating Datalog with the host language, Crepe enables seamless cross-language function calls, significantly improving the speed, ease-of-use, and expressiveness of the language for integrated queries. We showcased the real-world usage of Crepe through various open-source projects and evaluated its performance, where it was measured to be faster than compiled Souffle for an embedded workload and very close to the performance of Datafrog, a low-level engine.

In Part III, we introduced Percival, a novel language and reactive notebook environment designed for data analysis and visualization using Datalog. This experimental application demonstrates the potential of Datalog in the domain of exploratory data analysis, offering a tangible, reproducible, and shareable platform for data-driven insights. While Percival remains a prototype with much room for improvement, it highlights the potential of Datalog for addressing a broader spectrum of applications.

Programming languages serve as a bridge between human thought and machines. Effective language design provides mental frameworks that allow programmers to organize their thoughts and focus on the relevant aspects of a problem. Computation is a universal concept, and well-designed tools can facilitate ambitious pursuits by drawing users into a creative flow, balancing both the expressiveness of human thought and the efficiency and robustness of machine execution.

Working on Crepe and Percival illuminates a multifaceted perspective on how people think when they write software with a logic programming language. By examining both the big picture and individual usage patterns, we can discern common themes and challenges in embedded language design. Crepe is already used in production and could benefit from additional features, such as aggregation and more advanced join optimization strategies. Percival, as a prototype, requires significant development to reach its full potential, but we are already able to see how Datalog can lend itself to exploratory data analysis tasks.

In conclusion, the design and implementation of embedded logic programming systems in this thesis has provided insights into the potential of language design for various domains. By pushing the boundaries of Datalog's applications, we hope to contribute to a broader, unified understanding of how people effectively communicate with computers, paving the way for future advancements in programming languages and tools that foster creativity, productivity, and innovation.

# BIBLIOGRAPHY


[1] Molham Aref, Balder ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L Veldhuizen, and Geoffrey Washburn. "Design and implementation of the LogicBlox system." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. Piscataway, NJ, USA: IEEE Press, 2015, pp. 1371–1382.

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

[3] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. "$D^3$ data-driven documents." In: *IEEE transactions on visualization and computer graphics* 17.12 (2011), pp. 2301–2309.

[4] Stefano Ceri, Georg Gottlob, Letizia Tanca, et al. "What you always wanted to know about Datalog(and never dared to ask)." In: *IEEE transactions on knowledge and data engineering* 1.1 (1989), pp. 146–166.

[5] Will Crichton, Scott Kovach, and Gleb Shevchuk. *Expressiveness Benchmark*. 2020. URL: https://willcrichton.net/expressiveness-benchmark (visited on 03/09/2023).

[6] Inc. GitHub. *QL language reference*. 2023. URL: https://codeql.github.com/docs/ql-language-reference/ (visited on 03/07/2023).

[7] Todd J Green, Shan Shan Huang, Boon Thau Loo, Wenchao Zhou, et al. "Datalog and recursive query processing." In: *Foundations and Trends® in Databases* 5.2 (2013), pp. 105–195.

[8] D. Richard Hipp. *About SQLite*. 2022. URL: https://www.sqlite.org/about.html (visited on 03/03/2023).

[9] Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. "A classification of object-relational impedance mismatch." In: *2009 First International Confernce on Advances in Databases, Knowledge, and Data Applications*. IEEE. 2009, pp. 36–43.

[10] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. "Soufflé: On synthesis of program analyzers." In: *International Conference on Computer Aided Verification*. Springer. Piscataway, NJ, USA: IEEE Press, 2016, pp. 422–430.

[11] Ben Laurie. "Project Oak: Control Data in Distributed Systems, Verify All The Things." In: *DeepSpec at PLDI*. 2019.

[12] Wes McKinney et al. "Data structures for statistical computing in Python." In: *Proceedings of the 9th Python in Science Conference*. Vol. 445. Austin, TX. Austin, TX, USA: SciPy, 2010, pp. 51–56.

[13] Wes McKinney et al. "pandas: a foundational Python library for data analysis and statistics." In: *Python for high performance and scientific computing* 14.9 (2011), pp. 1–9.

[14] Frank McSherry et al. *Datafrog*. 2018. URL: https://github.com/rust-lang/datafrog (visited on 03/08/2023).

[15] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. "Differential Dataflow." In: *CIDR*. 2013.

[16] Bruce Momjian. *PostgreSQL: introduction and concepts*. Vol. 192. Addison-Wesley New York, 2001.

[17] Bernadette M Randles, Irene V Pasquetto, Milena S Golshan, and Christine L Borgman. "Using the Jupyter notebook as a tool for open science: An empirical study." In: *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*. IEEE. Piscataway, NJ, USA: IEEE Press, 2017, pp. 1–2.

[18] Leonid Ryzhyk and Mihai Budiu. "Differential Datalog." In: *Datalog* 2 (2019), pp. 4–5.

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

[20] Simon Segars. *Arm Partners Have Shipped 200 Billion Chips*. 2021. URL: https://www.arm.com/blogs/blueprint/200bn-arm-chips (visited on 03/01/2023).

[21] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S Lam. "Distributed socialite: A datalog-based language for large-scale graph analysis." In: *Proceedings of the VLDB Endowment* 6.14 (2013), pp. 1906–1917.

[22] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. "Big data analytics with datalog queries on spark." In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pp. 1135–1149.

[23] Evgeny Skvortsov. *Logica: Modern logic programming*. 2020. URL: https://logica.dev/ (visited on 03/07/2023).

[24] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. "Incrementalizing lattice-based program analyses in Datalog." In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), pp. 1–29.

[25] David Tolnay. *Serde: Serialization framework for Rust*. 2016. URL: https://crates.io/crates/serde (visited on 03/03/2023).

[26] David Tolnay. *Rust Latam: procedural macros workshop*. 2019. URL: https://github.com/dtolnay/proc-macro-workshop (visited on 03/03/2023).

[27] Jeffrey D Ullman. "Bottom-up beats top-down for Datalog." In: *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 1989, pp. 140–149.

[28] Bret Victor. *Magic Ink: Information software and the graphical interface*. 2006. URL: http://worrydream.com/MagicInk/ (visited on 03/01/2023).

[29] Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. "Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines." In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1542–1553.

[30] Hadley Wickham, Romain François, Lionel Henry, and Kirill Müller. *dplyr: A Grammar of Data Manipulation*. R package version 0.7.6. CRAN. 2018. URL: https://cran.r-project.org/package=dplyr.

[31] Yingjun Wu. *Comparison between RisingWave and Materialize*. 2022. URL: https://github.com/risingwavelabs/risingwave/discussions/1736 (visited on 03/06/2023).

## COLOPHON

This document was typeset using the `classicthesis` package developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography *"The Elements of Typographic Style"*. `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

*Final Version* as of March 24, 2023 (v1.0).