Efficient Staged Datalog Evaluation

Anna Herlihy
LAMP; IC; EPFL

***Abstract*—The challenge of extracting meaning and programmatically reasoning over external code appears in many areas of computer science, including but not limited to compilers, and database systems that support user-defined-functions. One approach to extracting behavioral guarantees from code is to use static program analysis, which is the analysis of computer programs without running them. Datalog, a declarative logic programming language has been proposed as a concise and intuitive alternative to hand-crafting complex static analyses in traditional programming languages like C++.**

The goal of this project is to provide a shallow embedding of Datalog in Scala backed by an in-memory execution engine. The research goal is to determine if multi-stage programming can be used to implement a Datalog execution engine that is both optimizable and retains the semantics of Datalog at each stage.

I. Introduction

THE challenge of extracting facts or meaning from code surfaces in many areas of computer science. A primary example is compiler systems, which ingest code written in a source programming language and translate into a target programming language. To do safety checks, for example to ensure that all variables are initialized, the compiler needs to understand some part of what the code is doing without running it. Another example is in database systems: many database systems support user-defined-functions (UDFs), which are a mechanism for extending the functionality of the database system. UDFs allow users to define their own functions that are called within database queries. These functions can be simply passed directly on to a compiler and then executed, thus essentially treated like black boxes. However treatment of UDFs is not optimal because there are missed optimization opportunities: the database system has access to additional information, for example statistics about the data that the function is operating over, that could be used to apply more aggressive optimizations. An alternative approach to UDFs is to require users to write their functions to conform to a specific template, for example Spark user-defined aggregation functions [28]. These templates are how the system is able to understand what the code is doing and apply optimizations, but it requires the function be customized to that system. This forces a tradeoff between performance and modularity or reusability.

One approach to extracting behavioral guarantees from code is to use static program analysis, which is the analysis of computer programs without running them [24]. In the simplest example, static analysis can identify if the test of a branch statement will always be false, and thus the body of the statement can be removed. This analysis, called dead-code elimination [20], has the benefits of shrinking program size and simplifying execution. Testing, meaning running a program with sample inputs and measuring the response, can identify program errors, but cannot guarantee that no errors exist [7]: however certain kinds of static analysis can check all possible executions of a program and provide stronger guarantees of correctness. Static analysis can also be combined with runtime code instrumentation [16]. However program analyses written in general-purpose programming languages like C++ have high development and maintenance costs due to the complexity and interconnectedness of analysis programs [23]. Static analyses often require solving constraint systems with complex inter-dependencies, for example call graph construction and conditional constant propagation. Datalog [11] is a rule based declarative language based in logic programming that has been proposed as a concise, maintainable alternative for expressing static analyses [11], [25], [23], [33], [17].

Because Datalog is based in logic programming, it can express complex interdependencies and deeply nested mutual recursion with simple logic statements. Static analysis programs that take hundreds of lines of code to express in traditional programming languages can be expressed in just a few lines of Datalog [33] with comparable performance [25]. The problem of Datalog evaluation can be understood as a program synthesis problem: the Datalog program is a declarative program specification and Datalog execution engines generate programs that execute that specification. This is a similar architecture to just-in-time code generating database systems [19], [21], which use SQL as the declarative specification. However the current state-of-the-art for Datalog and SQL execution is mostly limited to code generation that does not use runtime information to further specialize code once it has been generated, so there is a missed opportunity for optimizations that may only become possible when runtime information becomes available. Multi-stage programming (MSP) [31] is a metaprogramming paradigm that divides compilation into a series of stages, which allows for type-safe runtime code generation.

The first phase of the project was to implement a Datalog engine in Scala as an interpreter-based solver based on the semi-naive algorithm. The second phase of the project would leverage the Scala 3 metaprogramming constructs to do partial evaluation analogous to traditional DBMS JIT code generation, via implementation of a staged interpreter. The first phase is complete and the currently unnamed Datalog engine supports embedded pure Datalog in Scala. The Datalog API is defined in a user-facing “DSL” library, which calls into an “ExecutionEngine” that implements either the naive or semi-naive algorithm using a partial evaluation to relational algebra. The relational algebra operates over a custom volcano[13] based data “StorageManager” that supports select-project-join-union queries over relations are stored in Scala Collections. Each layer of the Datalog engine is pluggable, so we anticipate alternative storage systems (for example, Spark RDDs instead of Scala collections) and alternative evaluation strategies (for example, code generated operators instead of iterator-based execution). The next phase of the project, i.e. applying staging to Datalog execution, is currently in progress.

This report will first describe what Datalog is, then how it is evaluated, then how that can potentially be improved upon. The sections are organized as follows: first we provide relevant background on Datalog and its uses in Section II. We then describe related work in Section III. Lastly, we discuss a programming paradigm called multi-stage programming in Section IV. In Section V we conclude with a system description, evaluation and a plan for how MSP can be applied to extend code generation in Datalog evaluation beyond the current state-of-the-art, and various potential use cases.

II. Datalog Background

In this section we provide relevant background on Datalog [11], [15].

A. Datalog Syntax

In the following subsection we will provide a formal definition of Datalog and the terminology most commonly used to describe Datalog programs.

Logic programs contain a finite set of *facts* and *rules*. Facts are assertions about the domain, and rules are statements which allow additional facts to be deducted from existing facts. For example, in a directed graph, a fact could represent the statement "an edge exists between node 'A' and node 'B'" where 'A' and 'B' are constants. A rule could represent the transitive closure, for example "a path exists from node X to node Y if there exists an edge from node X to some midpoint node Z, *and* a path exists from node Z to the destination node Y". Rules usually contain variables to allow for generality, in this example X, Y, and Z.

In the formalism of Datalog, facts and rules are represented as causes of the form:

$$L_0 : -L_1, ..., L_n$$

where $n \geq 0$ and each L_i is a *literal*. Each literal consists of a *predicate symbol* p_i and a series of 0 more more *terms* t_j:

$$p_i(t_1, ..., t_{j_i})$$

A term can be either a *constant* or a *variable*. The left-hand side of the clause is called the *head* and the conjunction of literals on the right-hand side is called the *body*. Rules are read as logical implications ("$head$ is true if $body_1$ AND $body_2$ is true"). Clauses with no body are implied to be true, and such are facts. To represent our directed graph example, we can represent the fact "an edge exists between node 'A' and node 'B'" as:

$$edge('A', 'B')$$

And the rule "a path exists from node X to Y if there is an direct edge from node X to Y" as:

$$path(X, Y) : -\ edge(X, Y)$$

To complete our transitive closure example, we can extend our definition of a path to:

$$path(X, Y) : -\ edge(X, Z),\ path(Z, Y)$$

We say that the *definition* of a predicate symbol p in a Datalog program P is the set of rules of P having p in the head literal. In this example our predicate symbols are "path" and "edge"; the strings 'A' and 'B' are constants; and the symbols X, Y, and Z are variables. It is important to note that in the literature, when referring to Datalog programs, literals are often called *atoms* and predicate symbols called *relations*. The set of facts are called the *Extensional Database (EDB)* and the Datalog program rules are called the *Intensional Database (IDB)*. *Base* predicate symbols can only appear in the body of rules, and *derived* predicate symbols cannot appear in the EDB and their definitions are stored in the IDB.

All atoms of the same relation must be the same arity, i.e. the number of terms must be the same. A *ground* atom does not contain any variables. A Datalog program P must meet two safety conditions that ensure that an infinite set of facts cannot be derived from the finite set of rules: First, every fact of P is ground, meaning all the atoms of that fact are ground. Second, each variable which occurs in the head of a rule, must also occur in the body of the same rule.

We define the *dependency graph* G_p of a Datalog program P as the directed graph where the set of vertices is the set of derived predicate symbols and an edge exists from derived predicate symbols p' to p if a rule exists where p' appears in the body and p appears in the head. This allows us to define a *recursive* Datalog program as a program where G_p is cyclic: a predicate symbol is *recursive* if it occurs within a cycle in G_p, and a two predicate symbols are *mutually recursive* if they occur in the same cycle. It is important to note that the transitive closure example described above cannot be expressed using only relational algebra and calculus [2] due to the recursive nature of the program. Conversely, there are expressions in the relational algebra that cannot be expressed by pure Datalog programs, namely anything with the set difference operator.

B. Translation of Datalog into Relational Algebra

In the following subsection we will describe an algorithm to evaluate Datalog programs, starting with the derivation of relational algebra expressions for each derived predicate symbol p of P in a non-recursive program.

If we have a non-recursive program, then we know our dependency graph G_p does not contain any cycles and it is possible to do a topological sorting of the graph, so that an ordering is produced $p_1...p_n$ such that if there is an edge $p_i \rightarrow p_j$ then p_i comes before p_j. The relational algebra expressions are computed following this order.

For a Datalog rule r of the form:

$$p(t_1, ...t_m) : -R_1, ..., R_n$$

where R_i is the relation for each atom in position i.

The relational algebra expression for a derived predicate p is the union of the relational algebra expressions derived for each rule having p in the head. The relational algebra expression, expressed as $EvalRule(r, R_1, ...R_n)$, is derived as follows:

$$E = \sigma_F(R_1 \times ... \times R_n)$$

where σ is the *selection* operator:

$$\sigma_F(R) = \{t \mid t \in R \wedge F(t)\}$$

where F is a logical expression using $\wedge$, $\neg$, $\vee$, and atomic expressions of the form E_1 *op* E_2 where *op* is a comparison and E_1 and E_2 are constants or symbols. $F(t)$ represents the boolean value given by evaluating expression F over tuple t.

The conjunction of conditions F is obtained by comparing positions of terms in each relation that are the same variable, or for constants the position i. So if the body of R is

$$p_1(X, y),\ p_2(Y, Z, Z),\ p_3(X,\ 'a')$$

then, with position is written as $i,

$$F = (\$1 = \$6 \wedge \$2 = \$3 \wedge \$4 = \$5 \wedge \$7 =\ 'a')$$

We can then define $EvalRule$ as

$$EvalRule(r, R_1, ...R_n) = \pi_V(E)$$

where π is the *projection* operator:

$$\pi_V(R) = \{t[V] \mid t \in R\}$$

and V is a sequence of attributes to be retained in the result. V has the arity of the head predicate symbol p. So if the term at position i in the head is variable X, then V at position i will be one of the positions where X appears in the body of the rule. For example, if the body of r is the same as the example in step 1 and the head is defined as $p(X, Z)$ then the resulting expression is

$$\pi_{\$1,\$4}(E)$$

Each occurrence of a derived relation in E can be substituted with the corresponding relational algebra expression. Because we are evaluating predicate symbols in a topological order, we will have a relational algebra expression for each predicate symbol in the body of the current rule being applied. We can define $Eval$ as the union of each $EvalRule$ expression.

Intuitively, we can see that body atoms with common variables are derived as join operations, while the head is derived as projection operations.

C. Evaluation of Datalog Programs

In the following subsection we will use the algorithm described in the previous subsection to evaluate possibly recursive Datalog programs using an approach known as the *semi-naive* algorithm, or "bottom-up" evaluation. While alternative evaluation strategies exist, it has been shown that the semi-naive evaluation is as efficient or better than the alternative approaches [32] and modern Datalog engines, including the one discussed in Section III, base their architecture on the semi-naive relational algebra approach.

The intuition behind Datalog evaluation is that we apply rules to facts iteratively until there are no more facts to be derived, i.e. the evaluation has reached a fixed point. More specifically, the Gauss-Seidel method iteratively computes $Eval$ for each derived predicate symbol until an iteration

Algorithm 1 Naive-Evaluation

```
1: for $i := 1$ to $m$ do
2:     $P_i := \emptyset$;
3: repeat
4:     for $i := 1$ to $m$ do
5:         $P'_i := P_i$;
6:     for $i := 1$ to $m$ do
7:         $P_i := Eval(p_i, R_1, \dots, R_k, P'_1, \dots, P'_m)$;
8: until $P_i = P'_i$ for all $1 \leq i \leq m$
9: return $P_1, \dots, P_m$
```

Fig. 1: Pseudocode for the naive algorithm using $EvalRule$

Algorithm 2 Seminaive-Evaluation

```
1: for $i := 1$ to $m$ do
2:     $\Delta P_i := Eval(p_i, R_1, \dots, R_k, \emptyset, \dots, \emptyset)$;
3:     $P_i := \Delta P_i$;
4: repeat
5:     for $i := 1$ to $m$ do
6:         $\Delta P'_i := \Delta P_i$;
7:     for $i := 1$ to $m$ do
8:         $\Delta P_i := Eval\text{-}incr(p_i, R_1, \dots, R_k, P_1, \dots, P_m, \Delta P'_1, \dots, \Delta P'_m)$;
9:         $\Delta P_i := \Delta P_i - P_i$;
10:    for $i := 1$ to $m$ do
11:        $P_i := P_i \cup \Delta P_i$;
12: until $\Delta P_i = \emptyset$ for all $1 \leq i \leq m$
13: return $P_1, \dots, P_m$
```

Fig. 2: Pseudocode for the semi-naive algorithm using $EvalRuleIncr$

passes where no new facts are added to the set. In the case where we have two strongly connected components in the dependency graph G_p, we can apply a sort using strongly-connected components as the graph edges. Figure 1 shows the pseudocode for the naive evaluation of a Datalog program using the $Eval$ operator.

We can observe that the naive algorithm recomputes all tuples computed in the previous iteration in each subsequent iteration. The semi-naive algorithm is an optimization of the Gauss-Seidel method and seeks to avoid this wasted computation by incrementally computing new rules. To do this, we supplement $EvalRule$ defined in the previous section with an incremental version, $EvalRuleIncr$. Thus we also have an $EvalIncr$ that operates similarly to $Eval$, except it uses $EvalRuleIncr$ instead of $EvalRule$. We now need to keep track of an "incremental" relation ΔR_i for each relation R.

$$EvalRuleIncr(r, R_1, ..., R_n, \Delta R_1, ..., \Delta R_n) =$$
$$\bigcup_{1 \leq i \leq n} EvalRule(r, R_1, ..., R_{i-1}, \Delta R_i, R_{i+1}, ..., R_n)$$

Using $EvalRule$ and $EvalRuleIncr$, we can define the semi-naive algorithm in Figure 2. The Datalog evaluation in this project implements the semi-naive evaluation described above.

D. Datalog Optimizations

Mapping Datalog to the algebraic formalism means that we are able to reuse classical database optimizations. Because of this advantage, we chose to implement our own storage layer instead of reusing an existing RDBMS at the expense of slightly longer development time. Currently, the datalog engine is a simple iterator-style DBMS without advanced optimizations. In this subsection we will briefly describe some of the optimizations that can be applied to the Datalog evaluation engine in future work.

Variable reduction and constant reduction: CERI et. al. propose pushing selection operators before other operations, for example joins, to reduce the size and therefore the cost of the operation in an approach commonly known in the database literature as "predicate pushdown".

Conjunctive query optimization: the relational algebra expressions generated by the semi-naive evaluation strategy are unions of conjunctive queries, e.g. they follow the "SPJU" pattern (select-project-join-union). There has been significant research into this optimization problem, with the goal of finding a semantically equivalent query (a "core") that minimizes the number of conjunctions (which also decreases the number of expensive joins). Many of these approaches use static analyses to identify equivalences [1].

Common subexpression analysis: If a system receives multiple queries simultaneously, it is advantageous to isolate common subexpressions and treat the execution of these expressions as a sharable resource. This is especially advantageous for Datalog evaluation applied to static analysis because sub-analyses can be easily composed by taking the union of their rules [23].

Cost modeling: The quantitative determination of costs associated to each operation allows the query optimizer to mathematically model query execution and optimize for the best plan.

Automatic index selection: The problem of selecting the correct index for a database query has been studied extensively [6]. Current research on auto-indexing Datalog evaluation engines builds on top of database system research [30].

Incremental evaluation and views: If there are computationally expensive queries that are deployed repeatedly, there may be performance benefits to storing the results of the query and using the difference between successive database states to reduce the cost of evaluating subsequent queries. There has been significant work into expanding incremental evaluation approaches to recursive Datalog queries [9], [34].

Integrity constraints: Additional semantic information about the fact database can be used to improve performance. Integrity constraints express properties about the database which can be used to answer particular queries without having to read the facts themselves. For example, if a flight record database has the integrity constraint that "all flights going to Lausanne land in Cointrin Airport", then the query "What airport does the flight UA956 New York to Lausanne go to" can be answered immediately.

III. Related Work

Datalog has recently been used for Java[27], Tensor Flow[22], Ethereum Virtual Machine[14], and AWS network analysis[4]. Further, Datalog is the logical foundation for Dedalus[3], a language designed to model distributed systems that the Bloom language is founded on. Bloom uses Datalog as a basis to formally guarantee consistency properties of distributed systems. The HYDRO language stack[5] takes this approach a step further and argues that the next programming paradigm for cloud computing should involve using verified lifting to translate user-written programs into a logic-based internal representation language that has roots in Datalog.

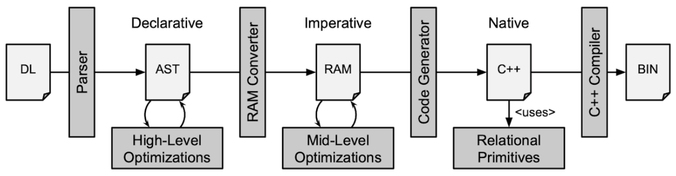

Fig. 3: Staged Compilation Framework for Synthesizing C++ Programs from a Datalog Specification

Yet there are still barriers to more widespread adoption of Datalog, in part because there are strong architectural boundaries between general purpose programming languages and external Datalog engines. Datalog engines also lack many benefits of general programming languages, like modularity and extensibility. Flix [23] is a declarative programming language that integrates first-class Datalog constraints into the language.

The current state-of-the-art for external Datalog evaluation is a system called Soufflé, which uses partial evaluation, compile-time code generation, and traditional relational database optimizations to synthesize efficient static analysis tools given a Datalog specification.

The key insight that Soufflé makes to surpass existing Datalog engines in performance is that the engine needs to be adapted, or *specialized*, to the input program specification. This insight leads to an understanding of the problem of efficient Datalog evaluation as a *program synthesis* problem: the goal of Soufflé is not to solve Datalog programs, the goal is automatically construct a program that satisfies an input specification. The input specification is the declarative Datalog program and the output is a heavily templatized C++ program that gets compiled into an executable by an optimizing C++ compiler. The reason why Soufflé produces heavily specialized programs is performance: the aim of Soufflé is to efficiently solve "real world" static analysis programs, defined as analyses on libraries that contain millions of variables, hundreds of attributes, and billions of tuples. Soufflé showed the generated programs can be just as fast and scalable as existing Datalog engines and hand-crafted tools.

A. *Soufflé Architecture*

The Soufflé compilation framework translates the input Datalog program into an executable in four intermediate stages, shown in Figure 3. In the first stage, the Datalog program is parsed into an abstract syntax tree (AST). At this stage the compiler performs semantic checks: correct relational symbol usage, variable type checks, and absence of cyclic negation (that may result in a Datalog program that never terminates). Logical optimizations can also be applied at this stage: including constant propagation and alias, rule, and relation elimination. In the next stage, the compiler performs the first partial evaluation of the input program to generate a "Relational Algebra Machine" (RAM) program. The RAM is an abstract machine used as a semantic model for evaluating the input Datalog program. The RAM intermediate representation consists of relational algebra operations (i.e. the relational expressions generated from Datalog clauses described in Section II-B), relation management operations to keep track of previous, current, and new facts (required by the semi-naive evaluation described in Section II-C), imperative constructs (i.e. statement composition and loop construction to express fixed-points), and parallelization operators similar to the Exchange operator in Volcano-based database systems [12]. The RAM is generated by reinterpreting the semi-naive evaluation described in Figure 2 on the abstract machine. Another way to describe this stage is to say that a *Futamura Projection* [10], is used to specialize the semi-naive evaluation for the facts provided by the Datalog program. Formally, applying a Futamura projection is applying partial evaluation to specialize an interpreter for given source code, yielding an executable program. In this case the result is executable on the RAM. During this phase various logical and physical optimizations can be applied to the RAM, including converting scans to range queries, index selection for nested-loop-join. In the third phase, a second partial evaluation is performed by generating imperative C++ code for each operator in the query. This code generation is similar to just-in-time code generation database systems [21], [19] that use code generation and specialization to improve query runtime. Soufflé additionally makes heavy use of C++ template metaprogramming, which allows certain static computations to be pushed from runtime to compile-time. The template metaprogramming is used to specialize the execution in two main ways: first with regards to data structures, which are in-memory. Soufflé will either select a B-tree or Trie based on the arity and available indexes for each relation. The reason Tries are introduced is due to the cost of reorganizing B-trees to execute relational operations in parallel: thus Soufflé uses Tries for relations with 1-2 attributes and the rest use B-trees. The second way template metaprogramming is used is to specialize the generated operator code for the relational operators. For example, the comparison operators are tuned to the arity and types of the relations and the search operators are tuned to the available indexes. The reason this specialization leads to more performant generated code is that there are fewer data-dependent control flow decisions, meaning the C++ compiler can apply more aggressive optimizations. In the last stage, an optimizing C++ compiler is applied to the generated C++ program and a native binary executable is produced.

The Datalog evaluation engine implement in this project is based on the Souffle system described above. An open question is if the system can take the staged approach a step further than Souffle's compile-time code specialization and potentially generate code at runtime.

IV. Multi-Stage Programming

In this section we discuss Scala multi-stage programming, a type of metaprogramming, and macros in the Scala language[29]. After introducing multi-stage programming, Section V explores how this approach can be applied to Datalog evaluation and applications.

A. Multi-stage programming in Scala 3

Metaprogramming is a programming technique in which programs can treat other programs as their data, sometimes as the input (for example analytical macros) or as the output (code generation). *Multi-stage programming* (MSP) [31] is a variety of metaprogramming in which compilation happens in a series of phases, starting at compile time and potentially continuing into multiple stages at runtime. Consider a just-in-time code generating database system that generates specialized code for the operators based on the query and knowledge of the data: then take it one step further and add specialization to the runtime of the query itself, based on information only available at runtime.

The multi-stage programming literature breaks staging into two fundamental operations, *quotes* and *splices*. Quotes essentially delay evaluation of an expression to a later phase. Delaying execution of code is commonly called *staging* in the literature. Splices compose and evaluate expressions, generating either expressions or another quote. MSP provides staged code generation so that code can be specialized at the level of abstraction appropriate for the optimization.

Quotes are written as `'{e}` for an expression `e`, and represent the typed abstract syntax tree for `e`. In the case that `e` is a type, then `'{e}` represents the non type-erased representation of the type. Splices are written as `${e}`, and evaluate the expression `e` and embed the result as an expression (respectively, type) in the enclosing program. `e` must yield a typed abstract syntax tree or type structure. Quotes and splices are duals of each other: for arbitrary expressions `e` we have

```
${'{e}} = e
'{${e}} = e
```

Given an expression `e` of type `T`, `'{e}` will have type `Expr[T]`. Given a type of type `T`, `'{T}` will have type `Type[T]`. So quoting is a function that takes expressions of type `T` to expressions of type `Expr[T]`, and takes types `T` to expressions of type `Type[T]`. Splicing is a function that takes expressions of type `Expr[T]` to expressions of type `T` and takes expressions of type `Type[T]` to type `T`. The Scala 3 Macro documentation [8] provides clear examples of quotes and splices in action.

The quote and splice operations can be combined with Scala 3's `inline` keyword, which guarantees inlining at type-checking time, in order to support a compile-time metaprogramming system. *Macro elaboration can be understood as the combination of a staging library and a quoted program.*

The stage that a quoted program is run is determined by the number of splice scopes and the number of quote scopes: if they are equal, the code is compiled and run as normal. If

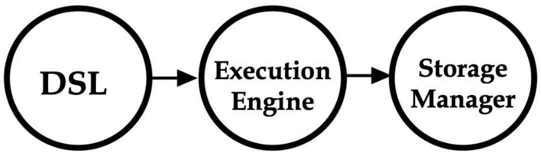

Fig. 4: Project Datalog Evaluation Engine Architecture

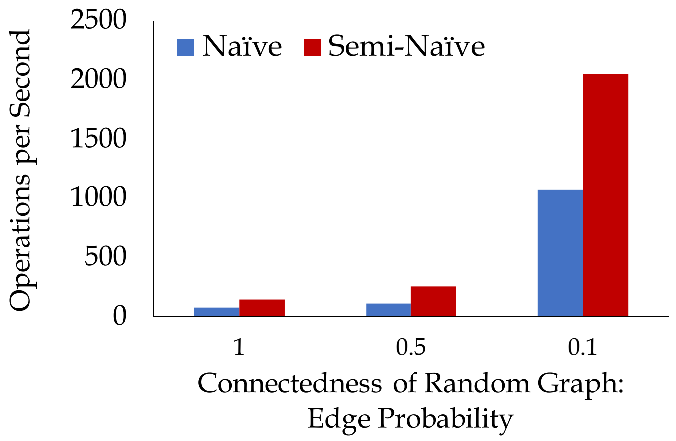

Fig. 5: Connectivity of a random graph. The graph was generated with an Erdős–Rényi model where for each pair of vertices, the probability of an edge existing is equal to *p*, the x-axis values. The semi-naive implementation is compared against the naive implementation for a series of 3 graph connectivity queries. Experiments were measured using the Java Microbenchmark Harness (JMH) [26] which is a benchmarking tool for JVM-based languages. Each experiment was measured after 15 warm-up runs and averaged over 10 iterations on a MacOS Big Sur 2.3 GHz Dual-Core Intel Core i5 8 GB 2133 MHz LPDDR3.

there is a top-level splice, the code is run at compile-time as a macro. If the number of quotes exceeds the number of splices, then the execution of the code deferred to a later stage. Each excess quote is an additional stage.

V. Scala Datalog Embedded DSL

The first phase of the project was to implement a Datalog engine in Scala as an interpreter-based solver based on the semi-naive algorithm. The next phase of the project would leverage the Scala 3 metaprogramming constructs to do partial evaluation analogous to traditional DBMS JIT code generation, via implementation of a staged interpreter.

The first phase is complete and the currently unnamed Datalog engine supports embedded pure Datalog in Scala. The system architecture is shown in Figure 4: The Datalog API is defined in a user-facing "DSL" library, which calls into an "ExecutionEngine" that implements either the naive or semi-naive algorithm using a partial evaluation to relational algebra. The comparison of naive and semi-naive can be seen in Figure 5. The relational algebra operates over a custom volcano[13] based data "StorageManager" that supports select-project-join-union queries over relations are stored in Scala Collections. Each layer of the Datalog engine is pluggable, so we anticipate alternative storage systems (for example, Spark RDDs instead of Scala collections) and alternative evaluation strategies (for example, code generated operators instead of iterator-based execution). The architecture was designed to be highly extensible at the expense of longer development time, but will hopefully serve as the foundation for future research directions. The second stage of the project, i.e. applying staging to Datalog execution, is currently in progress.

The next phase of the project is to answer the question of if runtime code generation via multi-stage programming can improve performance and enable novel optimizations, or if overheads from invoking a compiler at runtime are prohibitive. Due to timing constraints we were not able to provide an empirical answer to this question during the course of this semester but expect to have a working proof-of-concept imminently and experiments to follow.

Further future work is to extend the Datalog execution engine to virtually operate [18] over generated compiler artifacts, i.e. treating compiler-generated intermediate representations of code as the data layer in order to automatically and efficiently derive rules from code. The Scala 3 compiler generates a high-level interchange format, named TASTy (for Typed Abstract Syntax Trees), that contains information about the source code. This information includes syntactic structure, types, positions, and even documentation. As the Scala 3 compiler generates TASTy files with compilation, efficiently operating over TASTy files would enable advanced metaprogramming queries over Scala code. These advanced metaprogramming queries in both the Scala compiler as well as in database systems that execute UDFs. In the compiler, there are potential applications in initialization checking, dead code elimination, and incremental evaluation. In database systems, these queries over code will allow us to gain enough behavioral guarantees to apply novel or traditional DBMS optimizations to user-defined code.

VI. Acknowledgements

This project was undertaken as an EDIC semester project in the LAMP lab under the supervision of Martin Odersky. Thank you to Martin as well as Ondřej Lhoták, Nicolas Stucki, Guillaume Martres and the members of the LAMP and DIAS labs for their feedback.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. 01 1995.

[2] Alfred V. Aho and Jeffrey D. Ullman. Universality of data retrieval languages. In *Proceedings of the 6th ACM SIGACT-SIGPLAN POPL*, POPL '79, page 110–119, New York, NY, USA, 1979. ACM.

[3] Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. Dedalus: Datalog in time and space. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, pages 262–281, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[4] John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, Sean McLaughlin, Jason Reed, Neha Rungta, John Sizemore, Mark Stalzer, Preethi Srinivasan, Pavle Subotic, Carsten Varming, and Blake Whaley. *Reachability Analysis for AWS-Based Networks*, pages 231–241. 07 2019.

[5] Alvin Cheung, Natacha Crooks, Joseph M. Hellerstein, and Matthew Milano. New directions in cloud programming. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021.

[6] Douglas Comer. The difficulty of optimum index selection. *ACM Trans. Database Syst.*, 3(4):440–445, dec 1978.

[7] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured Programming*. Academic Press Ltd., GBR, 1972.

[8] Scala Documentation. Macros: Quotes and splices.

[9] Guozhu Dong and Jianwen Su. First-order incremental evaluation of datalog queries. In Catriel Beeri, Atsushi Ohori, and Dennis E. Shasha, editors, *Database Programming Languages (DBPL-4)*, pages 295–308, London, 1994. Springer London.

[10] Yoshihiko Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12:381–391, 1999.

[11] G. Gottlob, S. Ceri, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(01):146–166, jan 1989.

[12] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, page 102–111, New York, NY, USA, 1990. ACM.

[13] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6:120–135, 1994.

[14] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: Thorough, declarative decompilation of smart contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1176–1186, 2019.

[15] Sergio Greco and Cristian Molinaro. *Datalog and Logic Databases*. 2015.

[16] Anna Herlihy, Periklis Chrysogelos, and Anastasia Ailamaki. Boosting efficiency of external pipelines by blurring application boundaries. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, January 11-15, 2022, Online Proceedings*. www.cidrdb.org, 2022.

[17] L De Moura K Hoder, N Bjørner. Uz- an efficient engine for fixed points with constraints. In *Lecture Notes in Computer Science*, volume 6806, pages 457–462, 2011. 23rd International Conference on Computer Aided Verification, CAV 2011 ; Conference date: 01-07-2011.

[18] Manos Karpathiotakis, Ioannis Alagiannis, Thomas Heinis, Miguel Branco, and Anastasia Ailamaki. Just-in-time data virtualization: Lightweight data management with vida. 01 2015.

[19] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13):2209–2222, sep 2018.

[20] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN POPL*, POPL '73, page 194–206, New York, NY, USA, 1973. ACM.

[21] Andre Kohn, Viktor Leis, and Thomas Neumann. Adaptive execution of compiled queries. pages 197–208, 04 2018.

[22] Sifis Lagouvardos, Julian T Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. Static analysis of shape in tensorflow programs. In *ECOOP*, 2020.

[23] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. From datalog to flix: A declarative language for fixed points on lattices. *SIGPLAN Not.*, 51(6):194–208, jun 2016.

[24] Anders Møller and Michael I. Schwartzbach. Static program analysis, October 2018. Department of Computer Science, Aarhus University, `http://cs.au.dk/~amoeller/spa/`.

[25] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 196–206, New York, NY, USA, 2016. ACM.

[26] Aleksey Shipilev, Sergey Kuksenko, Anders Astrand, Staan Friberg, and Henrik Loef. Openjdk code tools: Jmh, 2022.

[27] Yannis Smaragdakis and Martin Bravenboer. Using datalog for fast and easy program analysis. In *Proceedings of the First International Conference on Datalog Reloaded*, Datalog'10, page 245–251, Berlin, Heidelberg, 2010. Springer-Verlag.

[28] Apache Spark. User defined aggregate functions (udafs).

[29] Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. A practical unification of multi-stage programming and macros. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2018, page 14–27, New York, NY, USA, 2018. Association for Computing Machinery.

[30] Pavle Subotić, Herbert Jordan, Lijun Chang, Alan Fekete, and Bernhard Scholz. Automatic index selection for large-scale datalog computation. *Proc. VLDB Endow.*, 12(2):141–153, oct 2018.

[31] Walid Taha. *A Gentle Introduction to Multi-Stage Programming, Part II*, page 260–290. Springer-Verlag, Berlin, Heidelberg, 2007.

[32] J. D. Ullman. Bottom-up beats top-down for datalog. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART PODS*, PODS '89, page 140–149, New York, NY, USA, 1989. ACM.

[33] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using datalog with binary decision diagrams for program analysis. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, page 97–118, Berlin, Heidelberg, 2005. Springer-Verlag.

[34] David Zhao, Pavle Subotic, Mukund Raghothaman, and Bernhard Scholz. *Towards Elastic Incrementalization for Datalog*. ACM, New York, NY, USA, 2021.