A Hybrid Approach to Logic Evaluation

LYNDON M. HENRY

B. Liberal Arts & Science (Hons.)

Supervisor: Prof. Bernhard Scholz
Associate Supervisor: Dr. Ying Zhou

A thesis submitted in fulfilment of
the requirements for the degree of
Master of Philosophy

School of Computer Science
The University of Sydney
Australia

10 May 2021

Authorship attribution statement

This thesis contains material published in "Efficient Sink-Reachability Analysis via Graph Reduction." by Dietrich *et al*. (see [19]). This is Section 4.2, Section 4.3, and Section 5.2. As a contributing author on this paper, I designed and implemented a system to evaluate the theoretical claims of [19], as well as performed all experiments and the analysis of results thereof included in the empirical evaluation of [19].

_____ Lyndon M. Henry 10th May 2021

Attestation of authorship attribution statement

In addition to the statements above, in cases where I am not the corresponding author of a published item, permission to include the published material has been granted by the corresponding author.

_____ Lyndon M. Henry 10th May 2021

As supervisor for the candidature upon which this thesis is based, I can confirm that the authorship attribution statements above are correct.

_____ Prof. Bernhard Scholz 10th May 2021

Statement of originality

This is to certify that to the best of my knowledge, the content of this thesis is my own work. This thesis has not been submitted for any degree or other purposes. I certify that the intellectual content of this thesis is the product of my own work and that all the assistance received in preparing this thesis and sources have been acknowledged.

_____ Lyndon M. Henry 10th May 2021

Abstract

In this thesis, we contribute the *hybrid approach* – a means of combining the practical advantages of feature-rich logic evaluation in the cloud, with the performance benefits of hand-written, optimized, efficient native code.

In the first part of our hybrid approach, we introduce a *cloud-based distribution* for logic programs, which may be deployed as a service, in standard cloud environments, across cheap commodity hardware. Modern systems are in the cloud; while distributed logic solvers exist, these systems are highly specialized, requiring expensive, resource intensive hardware infrastructures. Our original technique achieves a fully automatic synthesis of cloud infrastructure for logic programs, and includes a range of practical features not present in existing distributed logic solvers. We show that an implementation of the distribution scales effectively within real-world cloud environments, against a distribution over cores of the same machine. We show that our multi-node distribution may be effectively combined with existing multi-threaded techniques to mitigate the network communication cost incurred by distribution.

In the second part of our hybrid approach, we introduce *extra-logical algorithms*, to achieve performance for logic programs that would not be possible within a bottom-up logic evaluation. Modern systems must deliver high performance on big data; however, even the most powerful logic engines, distributed or otherwise, can be beaten by hand-written code on particular problems. We give a novel implementation of a system for the high-impact problem of sink-reachability, designed such that its algorithms may be used in logic programs. A thorough empirical evaluation, across a range of large-scale, real-world datasets, shows our system outperforms the current state of the art for the sink-reachability problem in all cases.

Our hybrid approach addresses the two major deficiencies of modern logic systems, providing a practical means of evaluating logic in distributed cloud-based environments, while offering performance gains for specific high-impact problems that would not be possible using logic programming alone.

Acknowledgments

Foremost, I would like to thank my supervisor Prof. Bernhard Scholz, whose good advice, measured patience, warm humor, and driven style have been of incalculable benefit. Additionally, I would like to thank Dr. Kamil Jezek, for providing technical expertise far above and beyond my own experience, and tireless assistance with running many of the experiments presented in this thesis. I would also like to thank Dr. Lijun Chang, Prof. Jens Dietrich, Long Qian, and Dr. Catherine McCartin for allowing me to collaborate on their research and publication, which forms a core foundation of the work presented in this thesis. And finally, I would like to express my gratitude to my parents and my sister, without whose tremendous support and unwavering encouragement none of this would be possible.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Logic programming is on the rise in both research and industry. Increasingly, logic programs are being effectively applied to solve real-world, large-scale problems.

Logic programs are declared as a specification, not an implementation. This gives logic programs a concise syntax, with less lines of code, and less bugs. The semantics are mathematical, and so reasoning on programs as mathematical abstractions is typical. This makes logic a powerful domain specific language.

For this work, we focus on a dialect in logic programming called *Datalog* [2, 11, 25, 24]. Datalog has had a turbulent past, which was tightly coupled with the rise of database systems, applications in artificial intelligence, and advances in the logic community. Datalog was appealing in communities such as Logic Programming, Database Systems, and Artificial Intelligence because of its simplicity, clean syntax, and semantics – this level of language expressiveness was sufficient for advancing the foundations and implementation of Datalog engines. A brief history of Datalog can be found in [11]. Datalog is a fragment of first order logic with recurrence.

A Datalog program consists of rules and facts. For example, the following program,

$$
\begin{array}{l}
E_{AB}. \\
E_{BC}. \\
R_{xy} \text{ :- } E_{xy}. \\
R_{xz} \text{ :- } E_{xy},\, R_{yz}.
\end{array}
$$

contains two relations, E and R. Relation E contains facts manifesting a set of edges of a graph. Note that the upper-case terms of E (i.e. A, B, C) are constants, while lower case terms (i.e. x, y, x) denote variables. Relation R captures the reachability of the graph defined by E. The reachability relation R is defined by two rules, in the form of horn clauses. The first rule says that if there exists an edge (x, y), then the reachability relation R will contain the edge. The first rule R_{xy} :- E_{xy}. is read as "(x, y) is in the reachability relation R only if (x, y) is an edge in in E". The second rule of the reachability relation R adds all transitive edges. This rule R_{xz} :- E_{xy}, R_{yz} is read as "(x, z) is in R only if (x, y) is an edge in E and (y, z) is already in the reachability relation R". Evaluating this program will compute all pairs of reachable vertices as the reachability relation R from the graph defined by E. After evaluation, the reachability relation R will contain the pairs (A, B), (B, C) and (A, C).

Since its inception, Datalog has seen continuous improvements. For example, the expressiveness of Datalog has been extended that include language elements such as negation,

aggregation, object-oriented extensions, advanced types such as records, and abstract data-types [2]. In this work we focus on Souffle [31, 61]. Souffle is a compiler for Datalog programs. Programs are translated to efficient C++ code, and executions may run with a high degree of parallelism on multicore machines.

Applications of Datalog are abundant. Datalog has been employed at NASA JPL, with the Semmle engine used to find a mission-critical bug during the 2012 Mars rover mission [62]. In the business domain, the LogicBlox Datalog system has been used to generate $150M of savings for a top 10 US retailer [44]. In software security, the Souffle Datalog engine has been used for security analysis of software defined networks numbering in the hundreds of thousands of edges, security analysis of the Ethereum blockchain over 6.6M smart contracts in under 10 hours, and, most notably, a points-to based security analysis of the entire OpenJDK in less than one minute [18].

Logic programming has many advantages in terms of syntax and semantics. However, Datalog is generally slower to execute than programs written in imperative or object-oriented languages for some applications. This is quite a natural observation since Datalog is declarative, i.e. the engine does not provide an execution order for statements and hence the machine needs to figure out efficient execution plans. This is an instance of the classic trade-off between the expressiveness and abstraction of high-level languages against the typically better performance of lower-level languages. Although modern Datalog engines such as Souffle's are fast, for some problems still more performance is required.

High-performance logic engines like Souffle are specialized systems, designed to run on resource-intensive hardware infrastructures, requiring dedicated massively multicore machines or expensive shared memory architectures. Distributed logic engines have been developed to improve the scalability of logic evaluation, both in research contexts using existing distributed computing platforms like Hadoop [4, 66, 64] Giraph [51], or Spark [67], and in industry, at Google [9] and Yahoo [13]. However, many omit features that make the language practical for real-world use cases – such as negation, or use of object types – preferring to sacrifice language expressiveness for performance in specific problem domains. These engines are also significantly restricted in their modularity and extensibility; they cannot be readily integrated with existing cloud infrastructures. Many also induce an overhead associated with simulating the logic evaluation within a distributed platform for which it was not designed, or suffer performance deficits due to the high cost of communicating information between machines.

The solution to the dual problems of obtaining a practical distributed Datalog evaluation, and achieving a performance on-par with state of the art systems is the contribution of this work.

First, we introduce a distributed evaluation machinery for Datalog in the cloud. By borrowing techniques from stream programming, we transform a logic evaluation into a stream program, consisting of a network of communicating actors that work independently from each other and communicate via channels. Using our distributed evaluation machinery, we can go beyond the resource boundaries of shared-memory multi-core machines. Our new approach provides a building block to scale up using a systematic way of mapping any Datalog program to a stream graph. Stream programs afford key advantages; they utilize a non-blocking communication

paradigm, giving high compute and network utilization, and achieve substantial parallelism, with evaluation distributed across concurrent nodes [50].

Results are streamed continuously between actors as they are computed, and program outputs produced gradually as they become available, rather than in bulk at the end of computation. Our implemented system can be deployed in a range of standard cloud environments, across cheap commodity hardware, and can interface with existing cloud systems, with logic as a service.

Second, for performance which cannot be obtained by logic programming alone, we describe how extra-logical predicates that encode algorithms can be incorporated into logic programs. Such algorithms are problem specific, hand-crafted implementations, and allow for the bottleneck of a logic program to be replaced by an optimized solution. In this work, we study how a type of graph reachability that is common in static program analysis – called sink reachability – can be incorporated via a Datalog engine. Sink reachability has many other application beyond points-to analysis, including analysis of semantic web graphs, XML document analysis, social network analysis, user access control network analysis, hyperlink analysis, and the analysis of relationships between genes. Much of our work on sink-reachability has been published in the paper of Dietrich *et al*. [19], on which I was a contributing author. In this thesis, we reproduce my contribution to that paper – being the implementation, and the experimental evaluation performed on it – but from the perspective of a system exposing algorithms for use as extra-logical predicates in Datalog programs. Our empirical results show state of the art performance for the high-impact problem of sink reachability, across large-scale, real-world datasets, drawn from points-to analysis, social network analysis, web graph analysis, and other application domains.

This combination of a cloud-based logic engine for achieving practical scalability, together with the use of extra-logical relations for achieving state of the art performance, is the *hybrid approach* – and the contribution of this work.

The thesis is organized as follows. In Chapter 3, we provide the theory of our streaming evaluation of logic, and the distribution that forms the first part of the hybrid approach. In Chapter 4, we present the design of our graph processing framework, a system implementing extra-logical algorithms as used in the second part of the hybrid approach. Chapter 5 is the empirical evaluation of both parts of the hybrid approach, evidencing results on each, with a focus on the problem of reachability. A review of the related work follows in Chapter 6, and then concluding remarks made in Chapter 7, including potential directions for future work.

CHAPTER 2

Background

2.1 Logic Programming

In this section, we provide background on the Datalog language. We show the syntax and semantics of the logic programming language, to serve as a foundation for the first part of our hybrid approach. We explain the seminaive algorithm, used to evaluate Datalog programs in the standard, non-distributed manner, to lay the groundwork for the modifications we make for evaluation of Datalog programs in the cloud.

2.1.1 Datalog

In Datalog, a program is written as a series of statements called *rules*. These rules operate over a set of input *facts* (called the EDB) to produce a set of output facts (called the IDB). An evaluation of a Datalog program P computes the output IDB facts from the input EDB facts.

For example, consider the problem of finding all paths in a directed graph. Representing this as a Datalog program, the edges E are the input EDB facts, the paths R are the output IDB facts. Such sets of facts in Datalog are called *relations*, and rules consist of logical statements that determine the content of these relations, based on the content of other relations. We can express the relationship between E and R as the following reachability program.

$$R_{xy} \text{ :- } E_{xy}.$$
$$R_{xz} \text{ :- } E_{xy}, R_{yz}.$$

Recall from the introduction that the first rule R_{xy} :- E_{xy}. is read as "(x, y) is in R only if (x, y) is an edge in in E" (for some values of the variables x and y), while the second rule R_{xz} :- E_{xy}, R_{yz} is read as "(x, z) is in R only if (x, y) is an edge in E and (y, z) is also in R" (for some values of x, y, z).

2.1.2 Evaluation of Datalog Programs

Facts are derived in a bottom-up fashion, that is, with new facts for the IDB (the output) derived from the EDB (the input). First, program rules are evaluated on the initial EDB facts to derive a first round of IDB facts. Then, the procedure continues as a series of iterations, in which rules are repeatedly evaluated at each iteration to derive new IDB facts, from facts derived in previous iterations. The iteration terminates upon reaching a fixpoint, where any

subsequent rule evaluation does not derive any new IDB facts. This procedure is captured by the seminaive evaluation algorithm, given in Algorithm 1. Our presentation closely follows the method in [2] and [51].

```
1 Δ ← T_P(∅)
2 I ← Δ
3 repeat
4 |   Δ_new ← T_P(Δ ∪ I) − I
5 |   I ← I ∪ Δ_new
6 |   Δ ← Δ_new
7 until Δ = ∅
```

Algorithm 1: Seminaive Evaluation

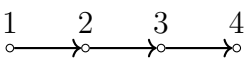
I is the set of all relations, called the *database instance*. Δ is the set of facts derived in the *previous* round, while Δ_{new} is the set of facts derived in the *current* round. T_P is the *immediate consequence operator* on the program P, and may be thought of as a function to evaluate all rules of P on a given database instance. More formally, given a database instance A, a tuple x is in the relation X in $T_p(A)$ iff either 1) $x \in X$ in the instance A or 2) there is a rule r in P, such that the body of r has a match in A, and x is the image of r's head under such a match.

The first line $\Delta \leftarrow T_P(\varnothing)$ populates Δ with the result of the immediate consequence operator T_P on an empty instance. This evaluates all rules of P that have only EDB relations in their bodies. In the next line, the statement $I \leftarrow \Delta$ adds Δ to the database instance I.

The algorithm then proceeds to evaluation of the fixpoint loop. $\Delta_{new} \leftarrow T_P(\Delta) - I$ computes Δ_{new} as the result of T_P on Δ, with any facts already in I excluded. $I \leftarrow I \cup \Delta_{new}$ merges Δ_{new} into the database instance I. $\Delta \leftarrow \Delta_{new}$ then assigns Δ_{new} to Δ, to have the new delta become the old delta in the next round. The loop terminates when Δ is empty, that is when T_P derives no new facts in the current round.

This algorithms called "seminaive" because, in contrast to the naive evaluation algorithm, which uses the full set of facts derived thus far to derive new facts for the current round, the seminaive algorithm uses only the newly derived facts (i.e. Δ) to compute the current round.

For example, consider the following directed graph.



As it relates to our earlier reachability program, the edge set of the graph would form the relation E, with $E = \{(1, 2), (2, 3), (3, 4)\}$. A trace of the seminaive evaluation of our earlier reachability example is given in Table 2.1.

Initially, in round 0, before the fixpoint loop but after the initialization of I and Δ, both $R \in I$ and $R \in \Delta$ have been populated with the relation E, while $R \in \Delta_{new}$ is empty. In round 1, after the first iteration of the fixpoint loop, Δ and Δ_{new} have new paths derived for R by the evaluation of the recursive rule R_{xz} :- E_{xy}, R_{yz}. . Here, $R \in I$ has all paths added previously

TABLE 2.1. Seminaive evaluation trace of left-linear reachability example.

	$R \in I$	$R \in \Delta$	$R \in \Delta_{new}$
0	$\{(1,2),(2,3),(3,4)\}$	$\{(1,2),(2,3),(3,4)\}$	$\varnothing$
1	$\{(1,2),(2,3),(3,4),(1,3),(2,4)\}$	$\{(1,3),(2,4)\}$	$\{(1,3),(2,4)\}$
2	$\{(1,2),(2,3),(3,4),(1,3),(2,4),(1,4)\}$	$\{(1,4)\}$	$\{(1,4)\}$
3	$\{(1,2),(2,3),(3,4),(1,3),(2,4),(1,4)\}$	$\varnothing$	$\varnothing$

plus the newly derived paths. Similarly, in round 2, after the second iteration of the loop, one new path has been discovered in the same manner as in round 1. In round 3, $R \in \Delta_{new}$ cannot be used in any rule evaluation to derive new paths, and so the algorithm terminates.

2.1.3 Negation in Datalog

The negation operator '$\neg$' is a useful feature of practical Datalog programs, however is omitted from almost all distributed evaluations. In the same way that the notation R_{xy} means "the tuple (x, y) is in the relation R" (for some value of x and y), the notation $\neg R_{xy}$ means "the tuple (x, y) is *not* in the relation R" Consider the earlier example, now modified with a negated literal of a new relation N.

$$N_x \text{ :- } A_x.$$
$$R_{xy} \text{ :- } \neg N_x,\, E_{xy}.$$
$$R_{xz} \text{ :- } \neg N_x,\, E_{xy},\, R_{yz}.$$

Here, the relation R is filtered by the relation N, as the first term x of any tuple added to R may not occur as the term of any (one-arity) tuple in N. As Datalog relations grow monotonically, that is, without removal of tuples that have already been added, negated relations must be evaluated strictly before their use in rule bodies. With reference to the example, the rule N_x :- A_x. for the relation N must be computed strictly before any rules for the relation R. If any rules for N were to be evaluated before any for R, then new tuples added to R may have as their first term x a value later added to N by the evaluation of the rule N_x :- A_x. . To account for this problem, an ordering must be imposed on the evaluation of relations for Datalog programs with negation.

2.1.4 Stratification of Datalog Programs

The technique of stratification imposes an order on relations, such that all negated relations are evaluated before the rules in which they occur negated. Stratification may be used to evaluate Datalog programs that include negation. Consider our earlier example of a Datalog program, with the negated relation N.

$$N_x \text{ :- } A_x.$$
$$R_{xy} \text{ :- } \neg N_x,\, E_{xy}.$$
$$R_{xz} \text{ :- } \neg N_x,\, E_{xy},\, R_{yz}.$$

A possible stratification is

$$S^1 = \{A, E\}, S^2 = \{N\}, S^3 = \{R\}$$

The evaluation proceeds over each stratum in sequence, evaluating all rules of the Datalog program for which a relation of the current stratum occurs as a rule head. As S^1 has relations A, E, which have no rules where they occur as a head, evaluation of S^1 is a no-op. Stratum S^2 has the relation N, and so the rule N_x :- A_x. is evaluated, adding all tuples from A to N. Finally, in S^3 the two rules for R are evaluated.

Stratification handles negation by realizing negated Datalog as a series of stratum or *semi-positive subprograms*, where negation is only allowed on relations occurring in rule bodies within each stratum. The correctness of the evaluation under stratification comes from the fact that each relation grows monotonically, without removal of tuples. In effect, each negated relation is treated as an output or IDB relation in the stratum in which it is computed (i.e. occurs as the head of rules), but as an input or EDB relation in the stratum in which it is used (i.e. occurs in rule bodies as a negated literal). More formally, a stratification is defined as follows.

Let $\mathbf{S} = \{S^1, S^2, \ldots, S^n\}$ be a partition of the relations $R^1, R^2, \ldots, R^m$ of a Datalog program P. Let S be the stratification index function, such that if $R^i \in S^j$ then $S(R^i) = j$.

- If a relation R^i occurs in the body of a rule for which R^j is the head, then $S(R^i) \leq S(R^j)$.
- If a relation R^i occurs *negated* in the body of a rule for which R^j is the head, then $S(R^i) < S(R^j)$.

The evaluation then proceeds to run the seminaive algorithm over each stratum in $\mathbf{S}$, according to the order of stratification index.

2.1.5 The Strongly-Connected Component Graph

A single Datalog program admits multiple stratifications. Recall our earlier example of negated reachability.

$$N_x \text{ :- } A_x.$$
$$R_{xy} \text{ :- } \neg N_x, E_{xy}.$$
$$R_{xz} \text{ :- } \neg N_x, E_{xy}, R_{yz}.$$

While previously we gave only one possible stratification of this program, there are actually many valid stratifications, including: $\{A\}\{E\}\{N\}\{R\}$, $\{A\}\{N\}\{E\}\{R\}$, $\{E\}\{A\}\{N\}\{R\}$, $\{A, E\}\{N\}\{R\}$, $\{A, N\}\{E\}\{R\}$, and, $\{A, E, N\}\{R\}$. The particular stratification used for the evaluation of Datalog programs is a standard construction via the strongly connected component (SCC) graph of the Datalog program [2]. The SCC graph is a directed acyclic graph which captures the dependencies involved in evaluation of Datalog relations, with vertices representing sets of relations, and an edge from a vertex u to v representing that the evaluation of all relations mapped to v depend on the evaluation of all relations mapped to vertex u. Each vertex of the SCC graph then corresponds exactly to one stratum in a stratification of the original Datalog program, and the order of the stratum is given by sorting the vertices of the SCC graph in a topological order. We see each stage of the SCC graph construction for the example program in Figure 2.1.

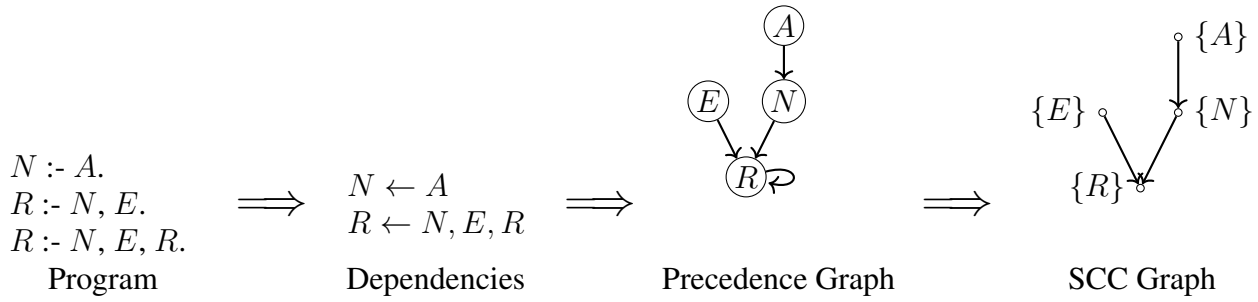


FIGURE 2.1. SCC Graph Construction

First, the program itself is given, with relation terms and negation omitted. Second, the relations of each of the rule heads are shown to depend on the relations in the rule bodies, using the $\leftarrow$ symbol to denote a dependency. Notice that the two rules for R have their dependency sets merged, with $R \leftarrow E, N$ given by the first rule R :- N, E. and $R \leftarrow E, N, R$ given by the second rule R :- N, E, . Third, the *precedence graph* is constructed for the Datalog program, with one vertex per relation, and one edge per dependency. Fourth and finally, the SCC graph is constructed from the precedence graphs. The difference between the precedence and SCC graphs is that (1) the precedence graph maps relations to single vertices, and, (2) there may be cycles as in the case of the self-loop on R, while in the SCC graph (1) vertices are mapped to sets of relations, and, (2) all vertices involved in a cycle are merged into a single vertex. The stratification is then given by a simple depth-first search over the SCC graph, which produces a topological order over the vertices.

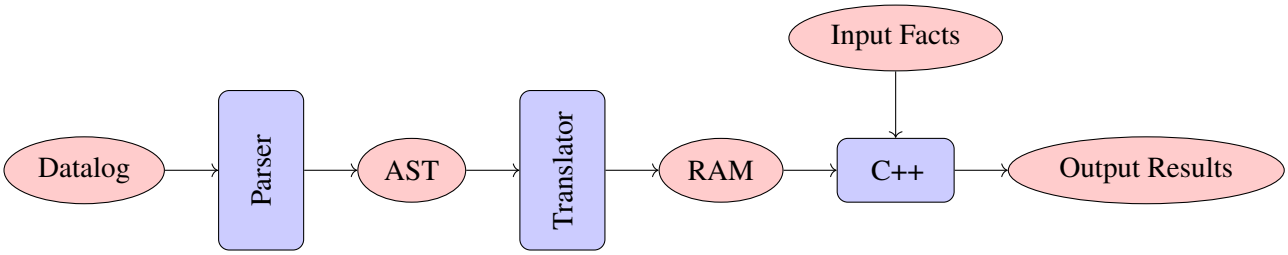
We now describe the general procedure more formally. Let $G = (V, E)$ be a directed graph, the dependency graph of P. For each relation R^i in P, there exists a vertex v_i in V. For each rule of P with a relation R^i as its head and a relation R^j occurring in its body, there exists an edge v_j, v_i in E (note that we allow self-loops, i.e. if $R^i = R^j$). Now, let $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, be the SCC graph of P. $\mathbf{V}$ is a set of sets of vertices in V such that initially, if $v_i \in V$ then $v_i \in \mathbf{v}_i$ and $\mathbf{v}_i \in \mathbf{V}$. The edges $\mathbf{E}$ are constructed initially such that if $(v_i, v_j) \in E$ then $(\mathbf{v}_i, \mathbf{v}_j) \in \mathbf{E}$. Cycles are then merged into strongly-connected components. For each pair of vertices $\mathbf{v}_i, \mathbf{v}_j$ in $\mathbf{V}$ with $i < j$, if $\mathbf{v}_i, \mathbf{v}_j$ are in a cycle, merge $\mathbf{v}_j$ into $\mathbf{v}_i$. In the case of recursive rules, the graph will contain such cycles. This merge removes $\mathbf{v}_j$ from the graph, while all edges having v_j as an endpoint now have v_i as their endpoint (with this applying to both inbound and outbound edges). When no more vertices remain to be merged, each set $\mathbf{v}_i \in \mathbf{V}$ corresponds to a stratum $S^i \in \mathbf{S}$. The edges of the SCC graph give an ordering over strata, such that if $(\mathbf{v}_i, \mathbf{v}_j) \in \mathbf{E}$ then $i < j$ for $S^i \in \mathbf{S}, S^j \in \mathbf{S}$. This gives a partial ordering over all strata of the program, as the edges of a directed acyclic graph. By sorting the vertices of the SCC graph in a topological order, we produce a total order over associated strata.

The typical whole program evaluation of a Datalog program involves stratification via the topological order of the SCC graph, and then executing the seminaive evaluation algorithm on each strata's semi-positive subprogram in the sequence of the topological order.

2.1.6 The Souffle Compiler and Interpreter for Datalog

Our system is implemented as an extension of the Souffle compiler and interpreter for Datalog programs. Souffle compiles Datalog programs, written in a terse declarative logic syntax, to imperative C++ code. The generated C++ uses optimized, custom built data structures, as well as various template programming techniques, resulting in efficient, high-performance compiled code. Souffle's dialect of Datalog supports negation, recursion, arithmetic, and aggregation – it has been shown to be Turing complete, by implementing a Turing machine within Souffle's Datalog dialect directly.

The Souffle compiler pipeline involves several translation phases, in its compilation of Datalog programs to C++ code. The pipeline is seen below, in a diagram reproduced from [29].



The first phase of compilation is the parser. An input Datalog program is translated into a data structure, called the abstract syntax tree (AST). The AST is run through a series of analysis stages, checking the validity of syntax and semantics. Then, a series of AST transformers are applied, which restructure the AST to provide high-level optimizations of the Datalog program, at the logic level.

The second compilation phase translates the AST to a program in Souffle's intermediate representation, called the Relational Algebra Machine (RAM) [61]. The RAM program describes the relational algebra operations involved in evaluation of the logic expressions that comprise the Datalog program. While Datalog is declarative, the RAM program is imperative, and captures the control flow of an evaluation. RAM programs are, like the AST, represented using a tree data structure. At the root of this tree is a node representing the program as a whole, whose children are nodes representing subroutines to be called in order. Each subroutine contains the various relational operations involved in rule evaluations, including searches over the tuples of relations, filtration of tuples, and insertions of newly discovered tuples to relations.

As with the AST, after construction, the RAM program is passed through a series of transformation stages. These apply low-level optimizations, such as index computation for searches over relations, and load-balancing for multithreaded executions [73].

As an example, we consider the RAM program corresponding to our earlier reachability program. Recall the first rule of the program

$$R_{xy} \text{ :- } E_{xy}.$$

This rule populates the relation R with every pair x, y in E. We read this as "x reaches y if there is an edge from x to y". The corresponding snippet of the RAM program is as follows.

```
1   ...
2   QUERY
3    IF (NOT (E = ∅))
4     FOR t0 IN E
5      PROJECT (t0.0, t0.1) INTO R
6   QUERY
7    FOR t0 IN R
8     PROJECT (t0.0, t0.1) INTO @delta\_R
9   ...
```

The snippet contains two `QUERY` blocks. The first `QUERY` checks if E is empty, and, if not, iterates over each tuple in E, adding it to R with the `PROJECT` operation. The second `QUERY` populates the R in Δ, as described earlier in our explanation of the seminaive algorithm, and written in the RAM program as `@delta_R`.

The second rule of the reachability program is as follows

$$R_{xz} \text{ :- } E_{xy}, R_{yz}.$$

This captures transitive reachability, and is read as as "x reaches z if there is an edge from x to some y and y reaches z". As this rule is recursive, the RAM program evaluates it within the fixpoint loop of the seminaive algorithm, as follows.

```
1   ...
2   LOOP
3    PARALLEL
4     QUERY
5      IF ((NOT (E = ∅)) AND (NOT (@delta\_R = ∅)))
6       FOR t0 IN E
7        FOR t1 IN @delta\_R ON INDEX t1.0 = t0.1
8         IF (NOT (t0.0,t1.1) ∈ R)
9          PROJECT (t0.0, t1.1) INTO @new\_R
10   END PARALLEL
11   EXIT (@new\_R = ∅)
12   QUERY
13    FOR t0 IN @new\_R
14     PROJECT (t0.0, t0.1) INTO R
15  SWAP (@delta\_R, @new\_R)
16  CLEAR @new\_R
17  END LOOP
18  ...
```

The RAM program contains a loop which is evaluated until the `EXIT` condition on line 11 is evaluated as true, under which circumstances the seminaive algorithm reaches a fixpoint, where no new tuples may be derived for any relation. Before the `EXIT` condition is a `QUERY` block, surrounded by `PARALLEL` and `END PARALLEL`. This means that the query will be

performed across multiple threads on partitions of the relations, where the number of threads and partitions is given at runtime. The parallel `QUERY` first checks that the relation E, and the R in Δ, written as `@delta_R`, are both non-empty. In accordance with the seminaive algorithm, `@delta_R` contains all newly discovered tuples in R from the previous iteration of the loop. By computing on `@delta_R` and not the full relation R, the seminaive algorithm avoids re-computing operations on tuples that have been added to R in prior iterations of the loop. A double-nested for loop follows, with outer iteration over tuples in `@delta_R`, and an inner iteration over tuples in E. For each tuple $(\texttt{t0.0}, \texttt{t0.1})$ in E and $(\texttt{t1.0}, \texttt{t1.1})$ in `@delta_R` where $\texttt{t0.1} = \texttt{t1.0}$ we add a tuple $(\texttt{t0.0}, \texttt{t1.1})$ to `@new_R` (i.e. R in Δ_{new}).

The `EXIT` condition checks if `@new_R` is empty, meaning that no new tuples have been derived by the parallel query block – when this condition is true, the algorithm immediately jumps to after the end of the loop. When the `EXIT` condition is false, the algorithm enters another `QUERY` block, and adds all new tuples in `@new_R` to R. Finally, `@new_R` is swapped with `@delta_R`, and `@new_R` cleared. In this way, the new information from the current round becomes the old information for the previous round in the next iteration of the fixpoint loop. The algorithm continues in this manner, iterating the loop, and monotonically increasing R until the `EXIT` condition is triggered.

In the final phase, C++ code is generated for the RAM program. Souffle also includes an interpreter mode of execution, which evaluates the RAM program directly and immediately – we do not use the interpreter in our work, and refer the reader to [61]. RAM statements are transformed into for-loops that directly implement the relational operations required in seminaive evaluation, relations become query-efficient data structures, and a simple CLI interface is generated to manage execution. The translation utilizes template meta-programming features of the C++ language to generate efficient Datalog-Enabled-Relations (DER). A DER data structure is specialized to the relation it represents [32, 33, 34, 52], with template parameters used to specify the DER implementation (B-tree, trie, etc.), shape (arity, type), and define an ordering used as a heuristic for relational operations. A DER permits efficient inserts, tests of membership, enumerations, and, highly efficient range queries for such heuristic orderings as specified by the template parameters. The C++ code is generated for all relations as DER's, as well as code for the operations defined in the RAM program, and compiled to an executable. When this executable is run, it evaluates the original Datalog program on a set of input fact files to produce a set of output result files.

2.1.7 Synthesis of Infrastructure

Souffle performs its translation by a projection of the given Datalog program into an instantiation of the seminaive evaluation algorithm itself, as realized in the RAM program. This is a sort of partial evaluation, where static information about the logic program, known at compile time, is used to construct a new program wherein which such static information is no longer passed at runtime, but hard-coded into the program itself. This process is known as a Futamura projection, and Souffle is said to synthesize imperative code from high-level logic [31].

Our extension continues in the same spirit as a program synthesis, however we now synthesize not only the imperative code, but the deployment of the logic program as a fully-fledged cloud system.

CHAPTER 3

Logic in the Cloud

3.1 Streaming Logic Evaluation

We now explain the first part of our hybrid approach, a technique for the distributed evaluation of Datalog programs in the cloud. Cloud computing is abundant, with most major technology companies providing cloud environments, including Amazon Web Services, Microsoft Azure, Google Cloud Platform, Alibaba Cloud, Oracle Cloud, IBM Cloud, and many others [21]. Distributed computing involves the use of multiple machines, communicating over a network, to perform some given task [38]. Components of this task are assigned as the workload of each individual machine, with the collective action of all machines used together to perform the task as a whole. Communication between machines may be achieved through a variety of methods, for example with all machines reading and writing to a single centralized data store (one to many communication), or by the passing of messages directly between individual machines for which they are relevant (one-to-one communication). Cloud computing, in distinction, is effectively a layer of abstraction over the physical architecture of distributed computing, where the specifics of the hardware infrastructure are hidden, and only a model of distribution exposed to the user [80]. For example, in a typical cloud deployment, a user does not provision physical machines, but requests an allocation of virtual machines or containerized environments, which may be located across cores of one physical machine, multiple physical machines, or a combination thereof.

This virtualization of resources affords advantages in both cost and scalability. Cloud computing offers the means of executing massively parallel computations on large volumes of data, using cheap commodity hardware. Cloud systems are designed to scale effectively to more runtime or memory intensive computations when provided with a respective increase in available resources. The advent of service-oriented architectures allows cloud technologies to be treated as modular and re-usable components, able to interface seamlessly with each other [55].

For these reasons, we here bring the advantages of the cloud to Datalog programs. We answer the primary research question of this first part of our hybrid approach: How can cloud computers be used for distributing a logic evaluation. We show how Datalog offers a natural distribution scheme for evaluation over nodes of a network. This affords the key advantage of an increased concurrency: multiple nodes operate in parallel to evaluate their own assigned subset of the total workload involved in a Datalog execution, rather than a single node being

used to evaluate each workload in a linear sequence. This allows performance to scale with an increase in the number of network nodes.

We will show how our extension provides a *synthesis of infrastructure*, whereby which a distribution is derived, fully automatically, from a logical specification. The network of the cloud environment is determined only by the logic program, with no change in how programs are written by the user (i.e. no message passing calls, parallel statements, locks, etc.). Our technique affords total backwards compatibility, where distributing any existing logic program written for Souffle is possible simply by compiling with our extension enabled. Using Souffle with our extension enabled defines and provisions the cloud environment based on the Datalog program – including the network nodes, the number of threads used by each, and the various network services required for the message-passing and coordination in the network of the distributed evaluation.

3.1.1 Distributing Logic via The Actor Model

Our technique involves distributing the evaluation of the seminaive algorithm across nodes of a network. We leverage the SCC graph construction, explained in the previous section, where vertices of the SCC graph are mapped to nodes of a network, and edges between vertices become communication channels over which network nodes pass messages.

We describe an abstraction of our distributed evaluation using a variant of Hewitt's actor model [27]. Actors provide a model of computation conceived of as a system of communicating processes (i.e. actors), and are defined by a few basic operations, given as follows. Actors can consume messages from a channel, produce messages to a channel, and evaluate some local computation. Such a local computation may make use of received messages, and/or result in new messages to be sent. Actors cannot access or modify any information outside of their own local state – that is, the local computation and local data received as messages.

While originally conceived of as a means to handle negation within Datalog programs, stratification enables thinking of a Datalog program as a set of smaller subprograms, where each subprogram corresponds to a stratum. As such, each stratum of a Datalog program may be evaluated independently from other stratum, with each stratum mapped to an actor, executing on a node of a network. This forms the method of our distributed technique, wherein which we map the stratification of a Datalog program to a system of actors.

The SCC graph construction captures dependencies between such strata in a partial order, with vertices corresponding to program strata, and edges corresponding to the dependencies themselves. The computation at each actor is the seminaive evaluation of the associated stratum, and there is exactly one actor per vertex of the SCC graph (equivalently, per stratum). In our distributed evaluation, the edges of the SCC graph define channels over which actors communicate the inputs and outputs of their strata evaluation, by message passing. Relations are communicated between actors, as messages over channels, and there is one channel associated with each relation. An actor consumes from the channels of the EDB relations of the subprogram it evaluates, and produces to the channels of the IDB relations of the subprogram it evaluates. Executing a distributed evaluation involves using all actors in the

network simultaneously to evaluate their assigned stratum. We now show how the seminaive evaluation can be modified to carry out this actor-based distribution.

3.1.2 Unstreamed Distributed Seminaive Evaluation

We modify the seminaive evaluation for the message passing context of our actor system, as shown in Algorithm 2.

$\Delta \leftarrow T_P(\varnothing)$
$\Delta \leftarrow \Delta \cup \textsc{Recv}(P)$
$I \leftarrow \Delta$
repeat
  $\Delta_{new} \leftarrow T_P(\Delta \cup I) - I$; $I \leftarrow I \cup I_{new}$; $\Delta \leftarrow \Delta_{new}$
until $\Delta = \varnothing$
$\textsc{Send}(P, I)$

Algorithm 2: Distributed Seminaive Evaluation

Calls to SEND and RECV are required within our actor-based distribution, as each actor does not have access to the results of previous strata's evaluation locally, and must obtain this information via messages. By abuse of notation, we use P here and henceforth to mean the subprogram of the stratum computed by the actor. During the call to $\textsc{Recv}(P)$, we iterate over each of the EDB relations R of the stratum subprogram P, and consume the full relation set of R from the channel (topic) for R. These messages are returned as a set of relations, which is then merged with the relation set of Δ. Similarly, in a call to $\textsc{Send}(P, I)$, we iterate over each of the IDB relations R of the stratum subprogram P, and produce the relation set of R in I to the channel for R. Overall, we receive EDB relations prior to computation, evaluate the fixpoint as per normal to obtain the IDB relations, then send off the IDB relations after computation has terminated.

The standard technique of evaluating program stratum in a topological order of their strongly connected component graph burdens evaluation with the unnecessary requirement of blocking behaviors. Each stratum must be fully evaluated before proceeding to evaluate its successor, regardless of whether the predecessor's evaluation is dependent on the successor. By operating only in the order of stratum dependencies, captured as the edges of the strongly connected component graph, multiple strata may be evaluated concurrently.

In our new evaluation, all actors compute their stratum simultaneously. Strata are evaluated once their local EDB becomes available from predecessor evaluations, with independent strata evaluation enabling concurrent computation. This avoids the need for a scheduler, while ensuring that computation of a stratum begins as soon as it is possible to do so.

The distribution technique affords a novel parallelism to Datalog evaluation. Prior approaches focus on a parallelization of program rules, either by a partition-based data parallelism over relations (e.g. as in [64]), or a task parallelism over the relational operations involved over rule evaluation (e.g. as in [3]). In contrast to this *fine-grained parallelism*, we offer a *course-grained parallelism* – operating not over program rules, but program stratum.

3.1.3 Streaming Seminaive Evaluation

In the standard, semi-naive evaluation of bottom-up logic programs, all rules of a stratum are evaluated before any output is produced. For this reason, the distribution exhibits widespread blocking behaviors. Each stratum is required to block computation, and wait to receive the full set of EDB relations before commencement of the fixpoint evaluation. Each stratum then computes the full IDB relation result, producing its result to other actors only at the end of this computation. Blocking is undesirable, resulting in poor resource utilization due to wait times. The solution is to move both consumption and production inside of the fixpoint loop. We provide an original generalization of the seminaive evaluation to simultaneously consume inputs, evaluate rules, and produce outputs, repeatedly, in an entirely non-blocking fashion. This enables relations to be streamed, with partial EDB received continuously over the course of evaluation, and partial IDB produced as results are computed. This brings high network and compute utilization, at the streams of our actor system continuously communicate data over the course of the evaluation, while the actors themselves are continuously evaluating their strata. By our stream-programming model, we achieve a non-blocking communication, with results streamed continuously between actors as they are computed, and program outputs produced gradually as they become available, rather than in bulk at the end of computation. This is useful if one wants only the first or first few results of a potentially long running query, as evaluation may be terminated after one (or few) iterations of the fixpoint loop. Streaming in this way trades a potentially higher computation load for a lower communication cost, by maximizing network utilization over the course of the evaluation, and keeping latency low.

We now give the modified streaming seminaive evaluation in Algorithm 9.

$\Delta \leftarrow T_P(\varnothing)$; $I \leftarrow \Delta$
$H \leftarrow \textsc{HaltOf}(EDB_P)$
repeat
  $\Delta \leftarrow \Delta \cup \textsc{Recv}(P)$
  $H \leftarrow H - \textsc{RecvHalt}(P)$
  $\Delta_{new} \leftarrow T_P(\Delta \cup I) - I$; $I \leftarrow I \cup I_{new}$; $\Delta \leftarrow \Delta_{new}$
  $\textsc{Send}(P, \Delta)$
until $\Delta = \varnothing$, $H = \varnothing$
$\textsc{SendHalt}(P)$

Algorithm 3: Streaming Seminaive Evaluation

The new HALTOF operation returns a set of special "halt" signals for each of the relations in the set passed to it. We initially create a set H, containing a halt signal for each EDB relation of P. This set H represents the halt signals yet to be received. When the algorithm enters the fixpoint loop, we RECV is called to consume any new EDB relations available at this time from their channel. RECV consumes all messages on the channel passed to it since its last call on that same channel. The $\textsc{RecvHalt}(P)$ operation then receives a set of any halt signals that have been produced for the EDB relations of P. Like RECV, RECVHALT consumes all halts on the channel passed to it since the last call on that channel. The set of received halts is removed from the set of all halt messages to be received H. The algorithm then evaluates

the fixpoint loop as previously, to derive Δ_{new}. Then, the SEND occurs – however instead of sending the full IDB relation set I only after termination of the fixpoint, we send the new IDB discovered at the current round Δ. Finally, the SENDHALT(P) operation then produces a set of halt messages for each of the IDB relation in P.

3.1.4 Streaming Seminaive Evaluation with Negation

The distribution we provide here is unique in how it handles negation. Most distributions of bottom-up logic evaluation do not include negation at all. Stream-programming solutions, in particular incremental and differential-dataflow systems handle negation in a manner requiring data to be streamed and processed multiple times [49] [50]. We offer a solution to the problem of distributing logic programs with negation that requires streaming data exactly once. The trade-off of our technique is that data streams involved in a negation in a rule must be fully received by their actor before the actor can begin computation, thus introducing blocking behaviors into an otherwise non-blocking communication.

While in traditional seminaive, the EDB relations are expected not to change, use of streaming violates this assumption. Each EDB relation is now effectively treated as an IDB relation, where consumption is treated in a manner similar to a rule evaluation. This requires a generalization of seminaive, where temporary rules must be created for each EDB relation as they are for each IDB relation.

As explained in the section on stratification, a negated relation must be evaluated before the rule body in which it occurs negated. For this reason, an actor in a distributed evaluation must block computation of its stratum until all negated relations it contains have been fully consumed. In this way, negation imposes blocking behaviors on computation of an actor's stratum. Our final modification of the seminaive algorithm accounts for this, by blocking subsequent computation until halts have been received for every negated relation. The modification is shown in Algorithm 4.

This algorithm adds a second loop before the main fixpoint loop. In this loop, we first initialize a set of halts for negated relations $H^{\neg}$. We then repeatedly receive any new messages to the Δ, and remove the set of received halts h from the set of all halts to be received H, and the set of negated halts to be received $H^{\neg}$. The loop terminates when the set of negated halts $H^{\neg}$ is empty, and from this point the algorithm continues as before.

3.1.5 Communication of Complex Data Types

Practical Datalog programs may make use of complex data types. This is enabled by allowing terms of relation tuples to be arbitrary objects (e.g. strings, records, etc.). Within the context of a non-distributed execution, such objects are stored in memory. Relations may store pointers to objects as terms of their tuples, in order to avoid copying an object redundantly. Within the context of our distributed execution, this approach is not possible. Each actor has only local state, and so if a tuple with a pointer to an object is received on some channel, that pointer will be invalid at the receiving actor. For this reason, many other distributed logic systems permit only basic datatypes, such as integers. These basic datatypes are passed by value, not

$\Delta \leftarrow T_P(\varnothing)$; $I \leftarrow \Delta$; $H \leftarrow \textsc{HaltOf}(EDB_P)$
$H^\neg \leftarrow \textsc{HaltOf}(EDB_P^\neg)$
repeat
  $\Delta \leftarrow \Delta \cup \textsc{Recv}(P)$
  $h \leftarrow \textsc{RecvHalt}(P)$
  $H = H - h$
  $H^\neg = H^\neg - h$
until $H^\neg = \varnothing$
repeat
  $\Delta \leftarrow \Delta \cup \textsc{Recv}(P)$; $H \leftarrow H - \textsc{RecvHalt}(P)$
  $\Delta_{new} \leftarrow T_P(\Delta \cup I) - I$; $I \leftarrow I \cup I_{new}$; $\Delta \leftarrow \Delta_{new}$
  $\textsc{Send}(P, \Delta)$
until $\Delta = \varnothing$, $H = \varnothing$
$\textsc{SendHalt}(P)$

Algorithm 4: Streaming Seminaive Evaluation with Negation

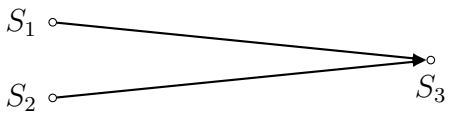
by reference – that is, a copy of the value itself is communicated, as opposed to a pointer to it. This leads to the same value being communicated multiple times, leading to sub-optimal communication overhead. We offer a novel solution for such complex datatypes, by passing serializations of heap objects, and mappings of pointers that refer to them between actors. This gives a means of communicating objects in a manner whereby which tuple terms that reference them are consistent across actors.

Our solution maintains a copy of each object that may be required in evaluation locally at each actor. The technique avoids use of shared memory, or a global state. Each actor stores a local mapping from an integer identifier to an object in memory. Relations then store these identifiers as terms of their tuples, enabling communication in the same manner as for numerically typed terms. At the time of rule evaluation, an identifier is resolved to the object copy local to the current actor.

We explain this process by means of an example. Consider the Datalog program given below.

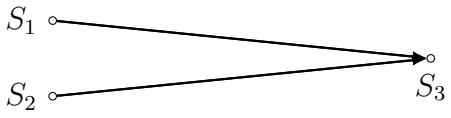
$$
\begin{aligned}
&A_{(\text{“}cat\text{''}, \text{“}dog\text{''})}.\\
&B_{(\text{“}dog\text{''}, \text{“}pig\text{''})}.\\
&C_{xy} \text{ :- } A_{xy}.\\
&C_{xy} \text{ :- } B_{xy}.
\end{aligned}
$$

The program consists of three relations, A, B, and C. The relation A is initialized with the tuple $(\text{“}cat\text{''}, \text{“}dog\text{''})$, while the relation B is initialized with the tuple $(\text{“}dog\text{''}, \text{“}pig\text{''})$. The two rules for C are trivial, adding tuples to C if they occur in either A or B. The SCC graph construction consists of three SCC's, $S_1 = \{A\}$, $S_2 = \{B\}$, and $S_3 = \{C\}$.



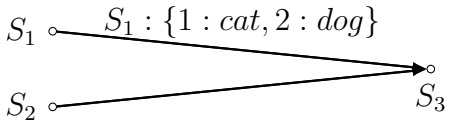
S_1		S_2	
R_1	T_1	R_2	T_2
$A : (1, 2)$	$1 : cat$ $2 : dog$	$B : (1 : 2)$	$1 : dog$ $2 : pig$

The SCC graph is given on the left in the figure above, while the initial state of the actors S_1 and S_2 are on the right. Here, T_1 and T_2 are the symbol tables for S_1 and S_2 respectively. On initialization, the relation A of S_1 is populated with the tuple $(\text{“}cat''$, $\text{“}dog'')$. However, as “cat” and “dog” are both strings, the relation A stores the pair of integers $(1, 2)$, which are the indices of “cat” and “dog” in the symbol table T_1. Additionally, relation B of S_2 is populated with the tuple $(\text{“}dog''$, $\text{“}pig'')$ on initialization. These are again stored as integers in the B relation of S_2, which now index strings in the symbol table T_2. Note that while “dog” occurs in both symbol tables T_1 and T_2, it has a different index in each – this shows how one string may be present at multiple actors, however be assigned a different local index. The third actor, S_3 has no initial tuples, and so the relation C and the symbol table T_3 are initially empty, as given below.



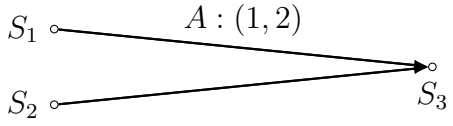
S_3		
R_3	T_3	M_3
$C : \emptyset$	$\emptyset$	$S_1 : \emptyset$ $S_2 : \emptyset$

In the table for S_3, we also see the mapping from remote to local indices M_3. As S_3 has one inbound edge from S_1 and one inbound edge from S_2 – shown in the SCC graph – the data structure M_3 has one mapping for each of S_1 and S_2. Any tuples received from S_1 (resp. S_2) will have their values replaced according to the mapping for S_1 (resp. S_2). We demonstrate this in action by proceeding through each step of the evaluation.



S_3		
R_3	T_3	M_3
$C : \emptyset$	$\emptyset$	$S_1 : \emptyset$ $S_2 : \emptyset$

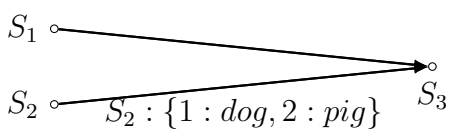
In the first step given above, the symbol table T_1 is propagated on the channel for S_1 to the actor S_3. The next step involves reception of these new symbols at S_3.



S_3

R_3	T_3	M_3
$C : \emptyset$	$1 : cat$	$S_1 : (1 : 1, 2 : 2)$
	$2 : dog$	$S_2 : \emptyset$

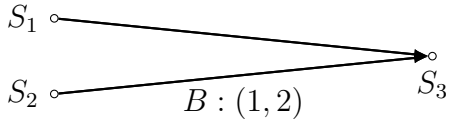
In this second step, symbols have been received at S_3 on the channel for S_1. The symbol table T_3 has been updated with the newly received symbols. The mapping for S_1 in M_3 has also been updated, we see that for S_1, the local and remote indices are the same at this time. That is, the remote index 1 at S_1 maps to the local index 1 at S_3, and the remote index 2 at S_1 maps to the local index 2 at S_2. Remote and local indices are not always the same, as we shall see in due course. Also at this time, the tuple $(1, 2)$ is sent by S_1 to S_3 on the channel for the relation A, and will be received by S_3 in the next step.



S_3

R_3	T_3	M_3
$C : (1, 2)$	$1 : cat$	$S_1 : (1 : 1, 2 : 2)$
	$2 : dog$	$S_2 : \emptyset$

In this third step, the tuple $(1, 2)$ has been received by S_3 on A's channel, and the relation C at S_3 has been updated according to the evaluation of the rule C_{xy} :- A_{xy}. . The symbol table for S_2 is also being propagated to S_3 – we note that this contains the duplicate string "dog", with a different symbol table index at the sender S_2 than at the receiver S_3.



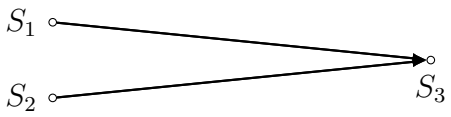
S_3

R_3	T_3	M_3
$C : (1, 2)$	$1 : cat$	$S_1 : (1 : 1, 2 : 2)$
	$2 : dog$	$S_2 : (1 : 2, 2 : 3)$
	$3 : pig$	

In this fourth step, we see the mapping at S_3 has been updated to resolve remote indices received from different sources to their correct local index. The remote indices 1 and 2 refer to different symbols at S_1 compared to S_2, and so the mapping must resolve the correct local index when tuples are received from S_1 or S_2 to ensure consistency. The string "dog" has the index of 2 at S_1, and when it is added for the first time to the symbol table of S_3, it retains this same index as S_3 as it has at S_1. The mapping of S_1 in M_3 shows this explicitly, as remote indices 1 and 2 map to local indices 1 and 2. However, the string "dog" has an index of 1 at S_2, and because it already exists in the symbol table at S_3, when it is received at S_3 the mapping for S_2 in M maps the remote index 1 to the local index 2. The string "pig" has an index of 2 at S_2, but when it arrives at S_3, the symbol table for S_3 already maps the index 2

to a different string. Hence "pig" is assigned the index 3 in the symbol table of S_3, and the mapping for S_2 in M resolves the remote index 2 – which refers to "pig" at S_2 – to the local index 3 – which refers to "pig" at S_3.

At this time also, the tuple $(1, 2)$ is being propagated on the channel of the relation B from S_2 to S_3, to be received in the next step.



S_3

R_3	T_3	M_3
$C : (1, 2), (2, 3)$	$1 : cat$	$S_1 : (1 : 1, 2 : 2)$
	$2 : dog$	$S_2 : (1 : 2, 2 : 3)$
	$3 : pig$	

The final step of evaluation shows result of receiving the tuple $(1, 2)$ for the relation B from S_2 to S_3. The values $(1, 2)$ are replaced according to the mapping for S_2's remote indices in M with their local indices, giving the new tuple $(2, 3)$. The rule C_{xy} :- B_{xy}. is then evaluated, adding the tuple $(2, 3)$ to C.

The general procedure for passing of complex object types such as strings is described as follows. Objects are communicated via special object channels, with one object channel per actor. Actors produce any newly derived or received objects to this channel, in a serialized form. Upon consuming a tuple with an object identifier as its term, an actor checks if that *remote* identifier has a corresponding *local* identifier in its object mapping. Each actor maintains a mapping from the remote identifiers present in received messages, to local identifiers present in the local object table. As remote actors may produce the same object but with different identifiers, there is one such remote to local identifier mapping per remote actor that an actor consumes from. If a local identifier is present in the local object mapping for a remote identifier in a newly received tuple, this remote identifier term is replaced by the local identifier. Otherwise, an actor will consume all objects on the channel of the remote actor, up to the remote identifier present in the received tuple. As identifiers increase in value monotonically, this results in new mappings from local to remote identifiers, and from local identifiers to objects, for each consumed identifier and object pair.

3.2 Extensions to Stream-based Evaluation

We have shown how Datalog programs may be evaluated in the cloud, utilizing the natural distribution offered by the SCC graph construction, and its abstraction via the actor model of computation. We have introduced a distributed variant of the seminaive evaluation algorithm for Datalog programs, with communication enabled by actors passing relations over channels as messages. We have then shown how this generalizes to a stream-based approach, allowing our distribution to operate in a non-blocking fashion, to produce results immediately and as soon as they are computed. Our technique provides a means for handling the often excluded, but practically necessary, feature of negation, and support for arbitrarily complex data types (e.g. string, records, etc.).

This next component of our technical contribution is concerned with extensions to the stream-based evaluation. The extensions operate at the logic level, and are not modifications of the underlying language itself, but instead are ways of constructing Datalog programs to increase the level of concurrency. Cloud environments are designed to scale the amount of work that can be performed in a given amount of time by varying the the amount of resources that are provisioned for a cloud deployment. This allows multiple machines to operate in parallel on a smaller subset of the workload, bringing higher performance.

The actor-based distribution we have introduced brings a *course-grained parallelism*, with evaluation of program strata parallelized across network nodes. Existing *fine-grained parallelism*, in which a evaluation of program rules is partitioned across multiple threads, is already implemented for our target system Souffle. Our course-grained parallelism is a form of task parallelism, operating at a higher level, and orthogonal to, existing fine-grained parallel approaches. We can combine our actor based, course-grained distribution, which splits the workload across nodes, with a data-parallel partition of relations across concurrent evaluations of a rule, on separate cores of the same node.

While highly advantageous, the combination of fine-grained parallelism with course-grained parallelism is still limited in the degree of concurrency possible. The fine-grained parallelism is bounded by the number of cores available on the network node an actor is executing on. The number of actors, and hence the number of network nodes, is a fixed constant, given by the number of vertices in the SCC graph, or, equivalently, the number of program strata. That is, we can increase the number of concurrently operating cores involved in an evaluation, but not the number of machines – the course-grained parallelism does not yet scale. In the extensions that follow, we show how to increase the number of actors involved in an evaluation, and so leverage the flexibility and scalability that cloud environments are designed for.

3.2.1 Streaming Optimization via Program Re-writes

The distribution of tasks within the course-grained parallelism is entirely dependent on the structure of the Datalog program. Each actor corresponds to one program strata, and each channel to one relation – the limit of course-grained parallelism is determined solely by the rules and relations of the program. The disadvantage of this is that one actor may dominate the

runtime of program evaluation, if one stratum is significantly more computationally intensive than the rest.

Such compute-heavy nodes perform disproportionate amounts of work, and lead to load balancing issues. Unfortunately, it is common that one or few compute-heavy strata dominate computation time – for example in computing a points-to analysis, runtime is dominated by the stratum computing the reachability relation of the variables to memory locations. This imposes a performance bottleneck. Course-grained parallelism offers no further concurrency here, short of combination with a fine-grained parallelism local at each actor. However, it is possible to leverage this property of structural dependence to introduce both additional task and data parallelism into a distribution, by re-writing the program itself. Such a re-writing involves "splitting" a Datalog program, a process in which multiple copies of the original program are concatenated together, each working on their own partition of the EDB. The results of all copies are then joined to obtain the IDB.

This is comparable to the data-parallelism achieved by a traditional partitioning of a relation over multiple threads, which each compute a copy of a rule on that partition, in parallel. However, our technique distributes data over a system of actors (i.e. nodes of a network), rather than over the cores of a single node.

It is also similar to systems like SociaLite [64], which employ a shard based approach where the first term of a tuple is used to partition a relation across a given number of shards. This partitioning is done either via hash or index, with shards split across nodes, then across the cores of each node. Our technique is different, however, in the sense that we partition relations by introducing entirely new relations and rules operating over them, moving the structure of the data distribution into the logic program itself. As a result, the technique may still be combined with existing fine-grained parallelism, and is not specific to the implementation details of our particular system.

3.2.2 Re-write of Non-linear Reachability

We demonstrate our re-writing technique by use of example, then give a description of the general method.

Consider the non-linear variant of our earlier reachability program.

$$R_{xy} \text{ :- } E_{xy}.$$
$$R_{xz} \text{ :- } R_{xy}, R_{yz}.$$

Here, the first rule is the same as for the linear variant of the Datalog program. The second rule is recursive, deriving new paths from only previously discovered paths, as opposed to from a combination of an edge and a previously discovered path. The second rule may be read as "if there is a path from x to y, and there is a path from y to z, then there is a path from x to z".

Consider a partition E^0, E^1 of E, such that $(x, y) \in E^1$ (resp. E^2) iff $(x, y) \in E$ and $x \bmod 2 = 0$ (resp. $a \bmod 2 = 1$). We may now "split" our example program in two by creating one copy to compute a relation R^0 on E^0, and another copy to compute a relation R^1 on E^1. The corresponding "join" involves merging the results of R^0 and R^1 with the original

relation to be computed, R. This split may be instrumented as the following re-write of the example program.

$$E_{xy} \text{ :- } E^0_{xy},\ x \bmod 2 = 0. \qquad E_{xy} \text{ :- } E^1_{xy},\ x \bmod 2 = 1.$$

$$\begin{array}{ll} R^0_{xy} \text{ :- } E^0_{xy}. & R^1_{xy} \text{ :- } E^1_{xy}. \\ R^0_{xz} \text{ :- } R^0_{xy},\ R^0_{yz}. & R^1_{xz} \text{ :- } R^1_{xy},\ R^1_{yz}. \\ R_{xy} \text{ :- } R^0_{xy}. & R_{xy} \text{ :- } R^1_{xy}. \end{array}$$

$$\begin{array}{l} R_{xy} \text{ :- } E_{xy}. \\ R_{xz} \text{ :- } R_{xy},\ R_{yz}. \end{array}$$

Note that we have removed the now redundant rule R_{xy} :- E_{xy}. in the re-write. This re-write involves a split of two clones, but the example program may be split into any number of such clones, demonstrated as follows

$$E_{xy} \text{ :- } E^0_{xy},\ x \bmod N = 0.\ \ldots \qquad E_{xy} \text{ :- } E^i_{xy},\ x \bmod N = i.\ \ldots \qquad E_{xy} \text{ :- } E^{N-1}_{xy},\ a \bmod N = N-1.$$

$$\begin{array}{lll} R^0_{xy} \text{ :- } E^0_{xy}. & R^i_{xy} \text{ :- } E^i_{xy}. & R^{N-1}_{xy} \text{ :- } E^{N-1}_{xy}. \\ R^0_{xz} \text{ :- } R^0_{xy},\ R^0_{yz}.\ \ldots & R^i_{xz} \text{ :- } R^i_{xy},\ R^i_{yz}.\ \ldots & R^{N-1}_{xz} \text{ :- } R^{N-1}_{xy},\ R^{N-1}_{yz}. \\ R_{xy} \text{ :- } R^0_{xy}.\ \ldots & R_{xy} \text{ :- } R^i_{xy}.\ \ldots & R_{xy} \text{ :- } R^{N-1}_{xy}. \end{array}$$

$$\begin{array}{l} R_{xy} \text{ :- } E_{xy}. \\ R_{xz} \text{ :- } R_{xy},\ R_{yz}. \end{array}$$

The split introduces $2N$ new relations, one for each of the N clones of E and R. Each of these relations are evaluated in their own separate stratum, hence adding $2N$ actors to the distributed evaluation. These actors may operate concurrently on their own partition of E and R.

3.2.3 Generalization of the Re-write Technique

In general, this splitting technique may be applied to any Datalog program. For a given split size N, we create N clones of the original program, with each clone of a relation R renamed to include their split index as R^i. For each new EDB relation clone E^i, we introduce a rule that populates E^i with the ith partition of the original relation E. For each IDB relation R, we introduce rules to join clones R^i of R with R. The cloned programs are then concatenated with the original program.

Next, every rule body literal involving negation of a cloned relation $\neg R^i$ is replaced with the negation of the original relation $\neg R$. Note that while there are no such negated relations in our example program presented here, this step is necessary for programs in general. Negated relations may not be split, for the same reason that they may not be streamed – the actor responsible for evaluating the rule with the negated relation requires the full negated relation for such an evaluation to be valid.

Finally, redundant rules introduced by the re-write are removed – these are rules which do not change the result of an evaluation if removed, and are specific to the Datalog program. In the example, this is the rule R_{xy} :- E_{xy}.

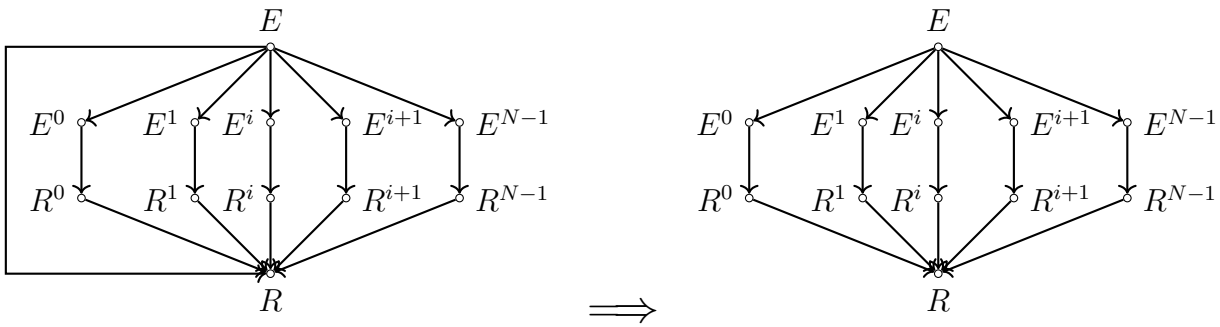
The correctness of the technique is guaranteed by the concatenation of the cloned programs with the original program. As the original program is still computed, each IDB relation will have *at least* the tuples that would be present in it if the re-write had not been performed. As all cloned IDB relations R^i are evaluated on a subset of the EDB, they each, and thereby their join as R, will have *no more* tuples than would be in R, had the re-write not been performed.

3.2.4 Redundant Rule Elimination

The final step, in which redundant rules are eliminated, must be performed with specific consideration of the Datalog program involved. If we were to remove the maximum amount of redundant rules, then we would remove all cloned rules from the program – this is because of the concatenation step, the original program is included in the re-written program, and only it's rules are strictly required to compute the original result. The re-written program is, in a sense, an over-approximation of the original program. By carefully selecting which redundant rules to remove, however, we may obtain dramatic changes in program structure.

Returning to our example, we will show the rule R_{xy} :- E_{xy} to be redundant in the final, re-written program. As the evaluation of this rule results in only $E \subseteq R$, we can show this rule to be redundant by deriving $E \subseteq R$ without use of this rule. By E's partition, $E = \bigcup_{0<i<N} E^i$. By the first rule of each split R^i_{xy} :- E^i_{xy}, we have that $E^i \subseteq R^i$, for all R^i. Hence $E \subseteq \bigcup_{0<i<N} R^i$. As each R^i is joined with R, $\bigcup_{0<i<N} R^i \subseteq R$. Therefore $E \subseteq R$, and so the original rule is redundant.

The difference in structure, and its advantage for the example program in particular, is most apparent from the SCC graph, given hence.



On the left is the SCC graph including the redundant rule, on the right is the SCC graph without it. There is an edge from E to R in the program with the redundant rule. As such, with the redundant rule, the actor for R may perform as much work as that of every split clone R^i, as it is possible for any tuple of R^i to be derived for R before some R^i does so and propagates it to R. Without the redundant rule, the distribution of work is spread evenly over

actors of the split relations. Each R^i is guaranteed to process tuples of E before they are propagated to R.

3.3 System Design and Implementation

In this section, we explain the design and implementation of our method for stream-based evaluation of Datalog programs in the cloud. Our implementation is based on a heavily modified version of the Souffle compiler and interpreter for Datalog programs. By augmenting the translation of Datalog programs to C++ that Souffle performs as its compilation, we replace the standard seminaive evaluation with our new distributed seminaive evaluation algorithm. The new C++ generated by this modified system includes various constructs for handling negation, program termination, and the communication of messages between actors.

3.3.0.1 Kafka

The communication of data within our distribution uses Kafka, a common industry standard technology for stream-based message passing in distributed environments, with Kafka systems supported by most cloud providers (Google, Microsoft, Amazon, etc). This allows logic programs to behave either as standalone executions, where program inputs and outputs are read to and written from files on disk, or as composable cloud services, where inputs and outputs for the logic program are communicated via Kafka streams. Kafka allows client programs to produce and consume messages to 'topics" (streams), with topics managed at a centralized broker server. These topics are the instrumentation of the relation channels. To handle the input and output facts, we replace the existing generated IO calls with two new actors – one to read all input files from disk and publish them on associated Kafka topics for input relations, and another to subscribe to Kafka topics associated with the output relations, and write those to output files.

3.3.1 Kafka Adapter API

To allow the program to act as a client to the Kafka broker, we developed a simple C++ adapter API over the commonly used librdkafka C library. The adapter API both extends and simplifies the librdkafka library. In the adapter API, there are various methods to attach and detach client consumers and producers to particular topics, and to send and receive messages via these producers and consumers.

It is possible to call the produce and consume methods of the API with different data types, including most numeric types, bytes, and C++11 string types, as well as vectors thereof. This makes sending and receiving Souffle types, in particular strings, vastly easier – as no custom serialization or deserialization logic is required at the site of the producing or consuming call.

The librdkafka library includes a relatively complicated polling mechanism, which requires a user to manually specify when the library is called to process outgoing messages in the producer's queue that have not yet been propagated to the Kafka broker. In librdkafka, the user is required to specify where in the code, and for how long (with a timeout), the librdkafka library should be polled. We use asynchronous polling for every produced messages during evaluation, with a final blocking poll at the call to a topic handle's destructor, which flushes the producer of that topic of all waiting messages.

The adapter API attaches logging to each of the methods it exposes, where a call to such a method will cause a log message to be issued on the log topic associated with the current execution. These log messages include various information on the operation for which they are omitted, including topic name, timestamp, and message payload size. This log mechanism is used in our empirical evaluation, to collect all experimental results over our system.

3.3.2 Kafka IO System

The generated code interacts with the Kafka API via Souffle's existing IO system. In the generated code, *load* and *store* statements are used to manage reading and writing relations. The semantics of these load and store statements is determined by the IO system, and it is possible to switch the back-end IO system used without altering the load and store statements of the generated code – e.g. instead of using the disk for IO, we may want to use an SQL database. Message-passing via Kafka replaces the existing file IO back-end with new Kafka IO classes to manage the loads and stores. Each Kafka IO class is instantiated with one relation, and so reads consume new facts the channel (Kafka topic) of that relation, while writes produce newly discovered facts since the previous writing call to their relation's associated topic. These Kafka IO classes also implement the logic for handling symbols, sending and receiving halt messages, connecting and disconnecting from Kafka topics, and ensuring that messages do not overwhelm the broker, by placing a max size of 100KB on all messages. All Kafka-related operations of the Kafka IO classes are done via calls to the previously discussed adapter API, these IO classes contain the logic of the communication protocol itself, not the specifics of the Kafka communication implementation. Within the generated code, only the placement and quantity of the load and store statements changes in the modified seminaive algorithm, not the actual statements themselves, and the program as a whole is configured to use the Kafka IO classes as the back-end.

3.3.3 Non-distributed Execution

The compiled executable produced by Souffle with the Kafka extension has several capabilities, controlled via command line options. The executable can create all Kafka topics required for a program evaluation, delete all Kafka topics of a program evaluation, or evaluate a given strata of the program as specified by its index. Other options exist for passing the number of threads used for multithreaded execution, and for executing the primary producer and primary consumer that manage reading and writing to disk (using stratum indices -2 and -3 respectively). Hence, the simplest means to fully evaluate a program involves the following steps.

(1) Start a Kafka broker.
(2) Use logic program compiled executable to create Kafka topics for program evaluation.
(3) For each program strata and the two strata, start a new (concurrent) process with a call to the executable and relevant stratum index.
(4) Wait for all output files to be written to disk.
(5) Use the executable to delete Kafka topics.

(6) End the Kafka broker.

This process is here described as a manual procedure, and results in all strata being evaluated on the same machine. We now describe our fully automatic cloud deployment.

3.3.4 Distributed Infrastructure

For the distributed deployment, the call to the Souffle compiler is replaced by a call to a wrapper script. This wrapper script takes the arguments for Souffle itself, together with some configuration options for the deployment. The wrapper script initially calls Souffle, passing the arguments, and resulting in the generation of a compiled executable. This executable is then uploaded to an S3 bucket. S3 is an online storage medium by Amazon Web Services. In our deployment, we use it as a cloud file store.

The wrapper script then generates a docker-compose.yml file, which defines the deployment. Docker is a technology that allows applications to run in isolated environments called containers, which are similar to independent virtual machines. Docker compose allows for the management of multiple Docker containers, according to a configuration file (a docker-compose.yml). A docker-compose.yml file defines which Docker containers to run, and which options to pass to them (either via arguments, or environment variables). Every docker-compose.yml file of our system defines several Kafka-related containers (including the Kafka broker, and a Zookeeper node). The docker-compose.yml file generated for a particular program has one container defined for each stratum of the program, plus one additional container to run the two strata concurrently. The definition of a container is called a Docker image, and there may be several containers using the same image, parameterized with different configuration. As such, the image used by all program containers, and image used by the container of the two primary actors, is the same for each container. When a docker-compose.yml file is deployed, all containers begin simultaneously, with access to the same network, and hence the Kafka broker. As our implementation uses Kubernetes to manage the docker deployment, containers are restarted in the event of failure (e.g. if the Kafka broker is not yet started and another container attempts a connection).

3.3.5 Distributed Execution

We now describe the image used by the program containers and primary containers, which manages the evaluation of the program. When a primary container starts, the compiled executable is first downloaded from the S3 bucket. A special Kafka topic is first created for logging the execution, this is the topic on which log messages are written, and is used to collect the results of our later empirical evaluation. Then, the compiled executable is downloaded from the S3 bucket. The primary container calls the executable with the command line option to create all Kafka topics related to program evaluation.

The primary container then creates another special topic, to signal the start of execution. This special start signal topic is used to coordinate the other actors, which wait indefinitely for this topic to exist before running their program. The primary container downloads input files from the S3 bucket, according to parameters passed as environment variables via the

docker-compose.yml file. Then, the primary container calls the executable on two concurrent processes, one for the primary producer, and one for the primary consumer. The primary producer reads all downloaded input files, and propagates them to the relevant topics. The primary consumer receives all messages published on topics of the output relations, and writes these to files. When the execution of both processes terminates, the output files are uploaded to the S3 bucket. The primary container then writes the entire log topic for this program evaluation to a file, and uploads this file to the S3 bucket. The existence of this log file in the S3 bucket signals the end of a program evaluation.

Within the container of each program strata, the procedure is different. Program containers download their executable from the S3 bucket, and then wait for the special start topic to be created by the primary container. When this topic exists, they call the compiled executable with the stratum index passed to the container via the docker-compose.yml file. The call evaluates the semi-positive subprogram of that strata, consuming and producing to various Kafka topics in the process.

All containers, both program and primary, enter an infinite loop at their last step. This prevents the Docker deployment environment (e.g. Kubernetes) from considering the node termination as a crash, and attempting to restart the container.

3.3.6 Remarks on Testing and Deployment

Note that only the docker-compose.yml file is required to deploy the system. All other assets are managed under the hood, with the S3 bucket used in place of a distributed file system. Docker compose is a common technology, and means for deploying stacks in this form via Kubernetes are supported by most major cloud providers (Amazon, Microsoft, Google, etc.). We have used Microsoft Azure as our chosen cloud platform, as a matter of prior experience and ease of use.

Additionally, the entire system itself is contained within the very image that is used by the actors. This means that one can build a program for deployment within a container of that image, and then run that deployment with stratum containers using copies of that same image – with the advantage of a dependency on only one Docker image. And as this image is published on a publicly accessible repository on the website Dockerhub, installing the system requires only one command to be run, with Docker as the only system dependency (i.e. `docker run lyndonmhenry/souffle-on-kafka`).

Souffle also includes a comprehensive test-suite, implemented mainly through a series of black-box tests (where only the input and output are tested). We have used this same test-suite to ensure the correctness of our system, utilizing a large number of tests under various configurations, including all configurations used in our experimental evaluation.

CHAPTER 4

Extra-logical Algorithms

4.1 Background

This chapter concerns the second part of our hybrid approach. Instead of targeting whole programs as in the first part of our hybrid approach – a technique that scales the performance of Datalog program in general – we target a specific, high-impact problem from the logic programming domain. We begin with a section on the background of the concepts and motivations of program analysis, and the role of Datalog in state of the art points-to analysis systems. We explain how the bottleneck of a points-to analysis is an instance of a particular problem of directed sink-reachability, and how by use of user-defined functors that call hand-crafted C++ code, we can effectively swap out this bottleneck for an optimized alternative. In this second part of our hybrid approach, however, we go beyond the confines of Datalog – adding characteristics to the language that would not be possible within a logic syntax and semantics.

4.1.1 Testing and Static Analysis

Testing verifies a subset of a program's behavior, by running a program or parts of it in particular states and on particular inputs, to produce particular changes in states and outputs. Tests pass when actual program states and outputs match those that are expected. Testing involves dynamic behaviors of programs, and has to do with what a program does at runtime. Ideally, behaviors that testing is designed to verify should be picked up when a program is built, that is, at compile time. Static program analysis offers a means to do this, by verification of a program itself, as opposed to its observed behavior.

One particularly successful form of static analysis is points-to analysis [18]. Points-to analysis can be used to detect a variety of bugs, including null pointers, tainted inputs, unreachable code paths, memory allocation errors, etc [68]. Points-to analysis, like other static analysis, works on an abstraction of a program. A program is considered as a set of variables, and objects (or locations on the heap). Variables are said to point to objects, and the analysis itself computes the points-to relation. Variable assignment to a new object is the simplest form of a statement inducing the addition of a heap location to a variable's points-to set. The analysis becomes increasingly more complicated when we consider that function calls, variable aliasing, object instantiation, fields of objects, and otherwise are all involved in how and whether a variable points-to an object.

The problem is in fact undecidable, as a variable's points-to set may be dependent on the result of a conditional statement, where the value of the conditional is determined by a program input. For example, in the snippet `int a = 0; if (x) a = 1; else a = 2;`, where `x` is determined by some argument to the program, `a` may have the value `1` or `2`. Therefore, points-to analysis aims for either a sound over-approximation, or a precise under-approximation, of the points-to sets of all programs. Additionally, the problem of distinguishing between variables on all code paths uniquely quickly becomes intractable for large code bases, as conditional statements and loops introduce a combinatorial blow-up. For this reason, points-to analyses are taxonimized further, by differing levels of sensitivity (e.g. call-site sensitive, object-sensitive, etc.).

4.1.2 Logic and Points-to Analysis

Logic programming, in particular Datalog, has found great success for points-to analysis [70]. The DOOP framework, which is written in Datalog, implements a whole-program points-to analysis, with full parameterization of context-sensitivity [68]. DOOP uses Soot to extract a set of input facts from a Java program. These facts represent all necessary information for the analysis, including assignments, fields, object instantiation, function calls, etc. The fact files are then used as the input EDB to a Datalog program, which computes a VARPOINTSTO relation on the program. The relation is derived recursively, using assignment statements, and following the call graph. The VARPOINTSTO relation is equivalent to computing the graph transitive closure, where variables and heap locations are vertices, while the call graph and assignment statements define an edge set. This transitive closure computation is the key performance bottleneck of DOOP, and points-to analysis more generally.

4.1.3 Directed Graph Reachability

Points-to analysis may be abstracted as a variant of the more general problem of directed graph reachability. The classic version of the problem is to answer the following query: for a directed graph $G = (V, E)$, given two vertices u and v in V, is there a directed path formed by the edges in E from u to v? The standard approach is to build the strongly-connected component (SCC) graph $G' = (V', E')$, using Tarjan's algorithm [2]. All vertices involved in a cycle in V are collapsed into a single vertex in V' (an SCC), with the edge set E' formed by the unique edges in E between each vertex in V'. The resulting graph G' is acyclic, and answering reachability queries on u and v in G may be done by answering the same query on $u' = SCC(u)$ and $v' = SCC(v)$ in G'. Iff there is a path (u, v) in G, then either there is a path (u', v') in G' or $u' = v'$.

There are two possibilities for answering queries. The first is to pre-compute the full transitive closure of all vertices in G', allowing for constant time queries, but with the trade-off of quadratic space. Alternatively, the quadratic space complexity may be avoided by using a standard search (i.e. BFS, DFS) to query reachability, however now with the trade-off that queries take $O(|V'| + |E'|)$ time. The two extremes of high space or high time complexity offered by this trade off make computing the reachability problem in this way intractable for

large graphs. Thus, a common approach employs attempting to find a sweet spot, by first pre-processing a graph to produce an efficient index structure, and then querying on this index structure to answer reachability queries.

The particular index we discuss here is the two-hop index of Cohen *et al.* [15] The two-hop index computes, for all vertices $v \in V$ of G, a set $L_{in}(v)$ and a set $L_{out}(v)$. To check reachability of a pair of vertices u, v in G, we check if $L_{out}(u) \cap L_{in}(v)$ – if there is such an intersection, u reaches v in G. Note that, in practice $L_{out}(v)$ and $L_{in}(v)$ for some v are typically small in size, making checking their intersection highly efficient. Typically, the smaller the size of the index – in terms of the sum of all sets $L_{in}(v)$ and $L_{out}(v)$ for all $v \in V$ – then the faster that reachability queries may be computed, and so the minimization of this value is a primary target and measure of performance.

Use of an index is more efficient than answering queries otherwise, when the space of the index is less than $O(|V|^2)$ (better than pre-computing the full transitive closure), the time of the index query is less than $O(|V| + |E|)$ (better than BFS/DFS), and the time of the index's construction is less than $O(|V| \times (|V| + |E|))$ (i.e. better than BFS/DFS on every vertex).

Unfortunately, in practice, according to Jin *et al.* most of the existing methods for reachability index construction reach a scalability bottleneck at one million vertices and/or edges [30]. Graphs formed by modern points-to analysis can number in the hundreds of millions or even billions of vertices and edges [18]. Existing approaches for computing reachability cannot scale effectively to large points-to graphs.

An attempt to solve this performance bottleneck is the notion of a reachability-preserving graph compression. In the same manner as the standard SCC transformation, we can apply other operations on a graph to reduce its size, while preserving the reachability relationship of its vertices. The work of Fan *et al.* introduces an equivalence reduction that preserves reachability in this way, where if vertices v, u have the same set of ancestors and descendants, then they are collapsed into the same vertex in the reduced graph [22]. This has been shown to achieve an average 95% compression factor for ten real-world graphs, with $|V| \in [6K, 2.4M]$ and $|E| \in [21K, 5M]$. However, the compression algorithm has $O(|V|(|V| + |E|)$ time, and $O(|V^2|)$ space, making it asymptotically no better than pre-computing an index via a direct BFS/DFS search and storing this explicitly as a set of reachable vertex pairs (i.e. the naive manner).

Zhou *et al.* demonstrate a combined strategy that can be scaled to large, real-world graphs [86]. Their technique involves first computing the transitive reduction of a graph, where all redundant transitive edges are removed to obtain the minimal unique graph with the same reachability as the original. The former equivalence reduction from [22] is then performed on this transitively-reduced graph, and the reachability index computed. While the transitive reduction has $O(V^3)$ time complexity in the worst case, the use of effective heuristics show the technique to be practical for real world graphs.

Although these approaches based on reachability-preserving compressions are effective, they still suffer from sub-optimal performance on large points-to graphs – we evidence this claim later on in the empirical evaluation in Chapter 5.

4.1.4 Sink Reachability

The solution to this scalability problem comes in considering points-to as a *restricted* variant of the reachability problem. In a points-to analysis, we are concerned with computing only the relationship between the set of variables and the set of objects/heap locations. This allows us to specialize the problem, and define a new graph, where V is the set of *source vertices*, representing variables, and S is the set of *sink vertices* representing objects/heap locations.

The sink-reachability problem takes as an input a *sink graph* $G = (V, S, E)$. In G, the set of vertices is divided into sink vertices S and non-sink vertices V assuming that the two vertex sets are disjoint (i.e., $V \cap S = \emptyset$).Note that, we assume that any successors/out-neighbors of sink vertices $s \in S$ are also contained in S, i.e., $N^+(s) \cap V = \emptyset$. The sink-reachability problem seeks for the *sink-reachability function* $r : V \to 2^S$ that captures, which sinks are reachable from a non-sink vertex in the sink graph. The notion of sink reachability may be seen as a refinement of the reachability relation, to answer particular queries more efficiently than in the general construction. In addition to points-to analysis, sink-reachability can be applied to the analysis of ID/IDREF, XLink links and entity references in XML documents, social networks, and large scale web graphs [56, 8]. For example, resource access controls on social networks may be modeled via sink reachability [1]. By treating resources as sink nodes and access rules as edges, $r(u)$ gives all resources accessible by a user u, while a reversal of edges obtains all users which may access a resource. Similarly, friend relationships between users can be represented as nodes and edges respectively, with sinks being used to model social media "influencers". The analysis of influencers is of high value for brand management, e-commerce, and social media marketing [1]. In the Semantic Web, graphs are found in the ID/IDREF links in XML documents or directly as RDF data. Efficiently performing various types of reachability query on such graph structured data is a topic of substantial investigation in the literature [79, 84, 12, 85]. Our notion of sink reachability is applicable to efficiently compute a range of queries in this domain.

A sink reachability problem arises when parsing structured data. Many expressive data formats have some notion of reference in order to avoid data redundancies. For instance, XML supports such references in the form of entity references. Those references can then be used by attackers to create malicious input that causes parsers to run out of time and/or memory as they have to evaluate an exponential number of paths, typically always resolving references to the same values via redundant paths. The classical vulnerability of this kind is CVE-2003-1564, better known as billion laughs. Similar attacks can be grafted for other data formats [20, 57]. The construction of the sink graph modeling relationships between references and values (with values being the sinks), and the use of effective algorithms to resolve references via sink reachability is a possible approach to avoid such attacks. In biology, transcriptional regulatory networks model relationships between genes as directed graphs. Nodes that are connected by an edge model where one gene regulates the activity of another. Efficiently answering reachability queries of such networks gives key insights into transitive regulatory relationships [46]. Sink reachability problem may be readily applied here also, to speed up queries involving designated subsets of genes, and those that are regulated by or regulate them.

Besides asking $r(u)$ for a query vertex u, another typical sink-reachability query might ask whether two non-sink vertices u and v share at least one sink vertex, i.e., whether $r(u) \cap r(v) \neq \emptyset$. Reachability queries of this particular type find applications in static program analysis (in particular points-to analysis) [58, 53, 68, 71], in analysis of percolation patterns in large real-world networks [82], and in logistics applications [39]. For example in points-to analysis, this can be used to answer aliasing queries i.e., two variables share at least one common sink vertex/object. This variation of the sink reachability problem has near-cubic running time, although algorithms with a sub-cubic worst-case time complexity exists [16, 72], they are not very practical nor efficient in practice. We note that the sink-reachability query asking whether $r(u) \cap r(v) \neq \emptyset$ can be answered by existing elementary reachability query processing techniques as follows: Given a sink graph $G = (V, S, E)$, we create its reverse graph $\overleftarrow{G} = (V', S', \overleftarrow{E})$, where every vertex $v \in V$ has a copy $v' \in V'$ and every vertex $s \in S$ has a copy $s' \in S'$. Then, we concatenate G and $\overleftarrow{G}$, denoted $G \oplus \overleftarrow{G}$, by adding a directed edge (s, s') for every $s \in S$. It is easy to verify that $r(u) \cap r(v) \neq \emptyset$ if and only if u can reach v' in $G \oplus \overleftarrow{G}$. Thus, all existing techniques for the elementary reachability problem can be applied here. In particular, the reachability preserving reduction techniques proposed in [87] can be applied to reduce the graph. However, this is not necessarily effective as it ignores the special properties of the sink-reachability function.

4.2 Sink-Reachability System

In our earlier chapter, we have seen how a generalized distribution of logic may be used to compute the graph transitive closure via a simple reachability program. In this way, we can distribute (via actors) and scale (via splitting) a Datalog program, increasing the available concurrency, and, ideally, it's performance. While this technique takes the top-down approach of providing a distribution for a feature-rich dialect of Datalog programs, the general nature of the solution induces a performance penalty. By extending the whole logic engine and evaluation scheme for all such Datalog programs, we disregard the specifics of particular target problems, an instance of a common trade off between generality of abstraction and performance in a specific domain. Indeed, it is typically possible to craft hand-written solutions to specific problems like transitive closure in a low-level language, where native code can make use of the assumptions of the problem domain.

In this section, we consider a means to extend Datalog programs for high-performance on specific problems via the use of purpose-built, optimized native code. We use transitive closure as our use case, and target the specific form of reachability required in a points-to analysis. We use specialized algorithms, realized in hand-crafted C++ code, to construct the points-to set externally to the logic program. It is then possible to embed this native code within a Datalog program, via the use of user-defined functors.

Our contribution in is the design and implementation of a sink-reachability preserving graph compression framework. This system has been used for the experimental evaluation of the paper in which the work on sink-reachability preserving graph compression has been originally presented (in [19]). It is designed to be extensible, for inclusion in a Souffle program or DOOP analysis (or any other system of Datalog permitting calls to native code). The system includes the current state of the art in reachability and sink-reachability preserving compression operations, having both the capability for sink-reachability specific compression, and the original implementation (the code itself, with the permission of the authors) of the transitive reduction, equivalence reduction, and two-hop indexing of [15] as part of our system.

4.2.1 User-defined Functors

The inclusion of native code within Souffle Datalog programs, in a manner that is consistent with both the logic syntax and semantics, is achieved by the use of user-defined functors, explained hence. Functors allow Datalog programs to make calls to functions that are otherwise impractical to evaluate within a Datalog program itself. A common example is the addition operation, i.e.many Datalog dialects allow two numbers to be added with the + functor. To define addition within a pure-Datalog language, it would be necessary to build up an arithmetic from scratch, by having a relation with every possible number, another for implementing a successor function, etc. Thus, we can include reachability as a user-defined functor, while preserving Datalog semantics.

The means of leveraging user-defined functors for incorporating an external points-to analysis is now demonstrated by use of example. Let E be an EDB edge relation of some Datalog

program P, and let R be the IDB path relation. Then, let f and g be two a user defined functors implementing the external sink-reachability program, and let G be a directed graph stored in memory. The f functor takes two arguments, and when called adds that pair of arguments to the edge set of G. In this way, we can call f with the arguments of each tuple in E to populate the edge set of G. Ideally, this could be done with a rule $f(x, y)$:- E_{xy}, however functors may not occur as the head of a Datalog rule, due to semantic constraints. Instead, we define a dummy relation F, and populate F by the rule F_{xy} :- $f(x, y)$, E_{xy}. Each call to f updates the state of the graph G internally, in memory, and always returns false so that the relation F does not consume memory. To obtain the IDB path relation R, we introduce a rule populating R via g. When g is first called, it computes the entire reachability index of G, and returns true if its first argument vertex reaches its second argument vertex, by querying on this index. Subsequent calls query the index, avoiding re-computation.

Again, it would be idea if a rule R_{xy} :- $g(x, y)$ were possible under Datalog semantics. Datalog variables must be *grounded*, that is, for any variable occurring in the head of a rule, that variable must occur as a term of a literal in the body of that rule. As g is strictly a functor, and not a relation, this requires explicitly creating a new vertex relation V, with the rule V_x :- $E_{x\_}$, $E_{\_x}$ defining V as having any endpoint of an edge in E (note that _ is a placeholder for any term). We may then use the rule R_{xy} :- V_x, V_y, $g(x, y)$. Note that we do not implement user-defined functors within a specific Souffle-dialect Datalog program, as this is a somewhat trivial detail to the research aims, and so left to future work.

4.2.2 Graph Processing Framework

The system implementing the sink-reachability computation is a graph processing framework, hand-written in efficient native C++ code. Graphs are processed by applying sequences of operators, where operators correspond to algorithms implemented by the system. While the system has a strong focus on linear-time sink-reachability graph compression operators, the design is extensible, and includes the linear equivalence relation of [22], the bottom-up transitive reduction of [86], and the two-hop index of [15]. These algorithms are implemented using exactly the same code as used in the original publications cited above, obtained with permission from the authors.

Use of the system involves applying operators in sequence to process input graphs, producing output graphs or queries. Users may specify graphs from files, and sequences of operators to run on them, via a command-line interface. Sets of operators may be run either repeatedly a given number of times, or until a first fixpoint is reached, where no operator in the given set may be applied to further reduce a graph, or, until the minimum possible operator sequence reaching a fixpoint is found. Operators sequences may also be specified to run before the fixpoint is reached (pre-processing), or immediately after a fixpoint is reached (post-processing).

All possible operations on the graph are implemented as operators, including queries such as computing the two-hop index, or counting all reachable vertex-sink pairs. This provides the powerful ability to perform complex computations and analysis on graphs, specified by a simple command-line interface. The software is designed to deliver empirical results on

graph operator applications, and includes capabilities to compute various statistics on graphs at each stage of operator application. The correctness of the implementation is continually verified at runtime, with operators required to ensure a set of preconditions over the state of the graph before execution, and assert a set of post-conditions on the graph after execution, thus providing the graph state for the precondition of the next operator in the sequence to be subsequently applied. As the system primarily deals in sink-reachability preserving compressions, the system includes the ability to check that sink-reachability is preserved after each operator application. Operator interfaces and graph data structures are designed for ease of re-use, and are readily adaptable as user-defined functions for inclusion in Souffle programs.

The design and implementation of the sink-reachability graph compression framework is comprehensively documented in the appendix of this work (see Chapter A). The available functionality of the sink-reachability preserving graph compression framework is captured in its command line interface (CLI), and is given in Section A1.4. A detailed treatment of the data structures and algorithms involved in the system, including detailed explanation of their method and computational complexity, is given in Section 24.

4.2.3 A Compositional Approach to Sink-Reachability

In this work, we present the design of our sink-reachability system. The remainder of this chapter primarily uses context from [19], the paper in which the work of this thesis is included, and on which the author of this thesis was a contributor. We omit the proofs included in [19], as these are not part of the contribution of this thesis. In this work, we introduce a highly scalable *sink-reachability preserving* graph reduction strategy to scale the sink-reachability problem to large graphs. The reduction strategy uses a *composition* framework consisting of condensation operators. We give four sink-reachability preserving condensation operators: strongly connected component condensation (operator S), condensation via dominators (operator D), and condensation via two special cases of module (operators $\mathsf{M_i}$ and $\mathsf{M_c}$). We devise algorithms to perform each condensation operator in linear time. We also investigate the properties of these four condensation operators, and how to compose them by noting that each condensation operator maps sink graphs to sink graphs.

4.2.4 Preliminary

We use $G = (V, E)$ to denote a directed graph consisting of a set V of vertices and a set E of edges. A directed edge from $u \in V$ to $v \in V$ is denoted by (u, v). For a vertex u, its set of out-neighbors is denoted by $N^+(u) = \{v \in V \mid (u, v) \in E\}$, and its set of in-neighbors is denoted by $N^-(u) = \{v \in V \mid (v, u) \in E\}$. The out-degree and in-degree of u are denoted by $d^+(u) = |N^+(u)|$ and $d^-(u) = |N^-(u)|$, respectively. A *path* in G from vertex $u \in V$ to vertex $v \in V$ is a sequence of vertices $(v_{i_1}, v_{i_2}, \ldots, v_{i_l})$ such that $u = v_{i_1}, v = v_{i_l}$, and $(v_{i_j}, v_{i_{j+1}}) \in E$ for all $1 \leq j < l$. We say that u can *reach* v in G, denoted $u \rightsquigarrow_G v$, if there is a path in G from u to v. We study sinks graph, a special type of directed graph.

DEFINITION 1. *A **sink graph**, denoted (V, S, E), is a directed graph $(V \cup S, E)$ where the set of vertices is partitioned into two disjoint subsets V (normal vertices) and S (sink vertices) such that there is no edge in G from S to V, i.e., $N^+(s) \cap V = \emptyset, \forall s \in S$.*

In this work, we focus on the problem of sink-reachability.

DEFINITION 2. *Given a sink graph $G = (V, S, E)$, the **sink-reachability** of a vertex $u \in V$ is the set of sink vertices that u can reach, denoted $r_G(u) = \{s \in S \mid u \rightsquigarrow_G s\}$.*

For example, Figure 4.1 shows a sink graph with $V = \{v_1, \ldots, v_{12}\}$ and $S = \{s_1, s_2, s_3\}$. The sink-reachability of v_1 is $r_G(v_1) = \{s_1\}$, and the sink-reachability of v_3 is $r_G(v_3) = \{s_2, s_3\}$. The relation $\simeq_r \subseteq V \times V$, where $u \simeq_r v$ iff (if and only if) $r_G(u) = r_G(v)$, is an equivalence

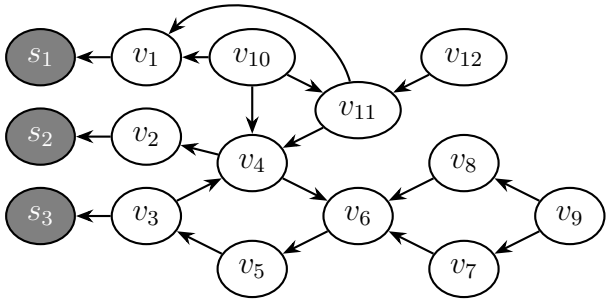


FIGURE 4.1. A toy sink graph

relation that is reflexive, symmetric, and transitive. Thus, if we contract each equivalence class of $\simeq_r$ into a super-vertex, the sink-reachability for all vertices is still preserved in the resulting graph, where the *equivalence class* of vertex $u \in V$ is $[u]_{\simeq_r} = \{v \in V \mid u \simeq_r v\}$. We define condensation for *any* equivalence relation as follows.

DEFINITION 3. *Given a sink graph $G = (V, S, E)$ and an equivalence relation $\simeq \subseteq V \times V$, the **condensation** of G induced by $\simeq$ is the triple $G_\simeq = (V_\simeq, S_\simeq, E_\simeq)$ such that*

- *$V_\simeq = \{[u]_\simeq \mid u \in V\}$ represents classes of vertices,*
- *$S_\simeq = S$, and*
- *$E_\simeq = \{([u]_\simeq, [v]_\simeq) \mid (u, v) \in E \wedge [u]_\simeq \neq [v]_\simeq\}$*

where self-loops and parallel edges are removed, and the equivalence relation $\simeq$ is extended to include S by defining $[s]_\simeq = \{s\}$ for each $s \in S$.

For example, Figure 4.2 shows the condensation of G induced by the equivalence relation that has two non-trivial equivalence classes: $\{v_{10}, v_{11}, v_{12}\}$ and $\{v_3, v_4, \ldots, v_9\}$. Note that $G_\simeq$ is also a sink graph. That is, *condensation maps sink graphs to sink graphs*.

DEFINITION 4. *Given a sink graph $G = (V, S, E)$ and an equivalence relation $\simeq \subseteq V \times V$, the condensation $G_\simeq$ is **sink-reachability preserving** if it preserves the sink-reachability for all vertices of V, i.e., $r_G(u) = r_{G_\simeq}([u]_\simeq), \forall u \in V$.*

As a condensation reduces the graph size (i.e., $|V_{\simeq}| \leq |V|$ and $|E_{\simeq}| \leq |E|$), we refer to a sink-reachability preserving condensation as a *sink-reachability preserving graph reduction*. The condensation $G_{\simeq_r}$ is the smallest (in terms of vertex number) sink-reachability preserving reduction we can obtain. Thus, we call the relation $\simeq_r$ the ***kernel equivalence relation*** and the condensation $G_{\simeq_r}$ the ***kernel condensation***. For example, the condensation in Figure 4.2

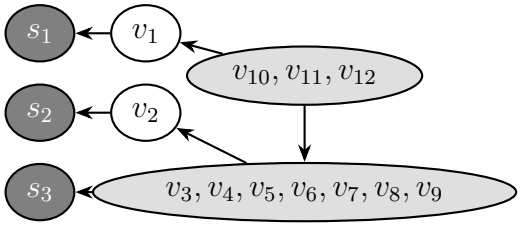


FIGURE 4.2. Kernel condensation of the graph in Figure 4.1

is the kernel condensation of the graph in Figure 4.1. However, directly computing the kernel condensation requires at least quadratic time in the worst case which is prohibitive for large graphs with millions of vertices. Thus, we resort to approximating the kernel condensation. We say that $G_{\simeq}$ ***under-approximates*** the kernel condensation $G_{\simeq_r}$ if $\simeq \subseteq \simeq_r$. It is easy to see that a condensation is sink-reachability preserving iff it under-approximates the kernel condensation.

Problem Statement. Given a sink graph $G = (V, S, E)$, we study the problem of sink-reachability preserving graph reduction, i.e., compute a condensation $G_{\simeq}$ that is small and under-approximates $G_{\simeq_r}$. Without loss of generality, we assume that the input sink graph G satisfies $r_G(u) \neq \emptyset$ for every $u \in V$; otherwise, all such vertices with $r_G(u) = \emptyset$ can be removed from G in a pre-processing step in linear time. Frequently used notations are summarized in Table 4.1.

TABLE 4.1. Frequently used notation

Notation	Description
(V, S, E)	Sink graph where $V \cap S = \emptyset$ and $N^+(s) \cap V = \emptyset, \forall s \in S$
$u \rightsquigarrow_G v$	u reach v in G, i.e., there is a path in G from u to v
$r_G(u)$	Sink reachability of u: $r_G(u) = \{s \in S \mid u \rightsquigarrow_G s\}$
$t(u)$	Topological number of vertex u in a DAG
$\simeq$	An equivalence relation $\simeq \subseteq V \times V$
$\simeq_r$	The kernel equivalence relation, where $u \simeq_r v$ iff $r_G(u) = r_G(v)$
$[u]_{\simeq}$	The equivalence class of u that is defined by $\simeq$
$G_{\simeq}$	sink reachability reduction w.r.t. $\simeq$
$\mathbb{C}$	A composition sequence of condensation operators

4.3 Composition of Sink-Reachability Operators

In this section, we propose a compositional approach to sink-reachability preserving graph reduction. We first investigate four linear-time condensation operators in Section 4.3.1, and then compose them in Section 4.3.2.

4.3.1 Linear-Time Condensation Operators

We study the condensations induced by SCC-, DOM-, IMOD-, and CMOD-equivalence relations in the following three subsections.

4.3.1.1 SCC-Condensation

Our first condensation is based on the concept of strongly connected component (SCC). An SCC of a directed graph is a maximal set of vertices such that every pair of its vertices can reach each other [17].

DEFINITION 5. *Given a sink graph $G = (V, S, E)$, two vertices $u, v \in V$ are said to be* SCC-***equivalent**, denoted $u \simeq_{\text{SCC}} v$, if they belong to the same SCC in G.*

The relation induced by SCC-equivalence is reflexive, symmetric, and transitive. Moreover, SCC-equivalent vertices are also equivalent in the kernel equivalence relation, i.e., $\simeq_{\text{SCC}} \subseteq \simeq_r$. Hence, the condensation induced by the SCC-equivalence relation, called SCC-condensation, is sink-reachability preserving. For the sink graph in Figure 4.1, the only non-trivial SCC is $\{v_3, v_4, v_5, v_6\}$, and its SCC-condensation is shown in Figure 4.3.

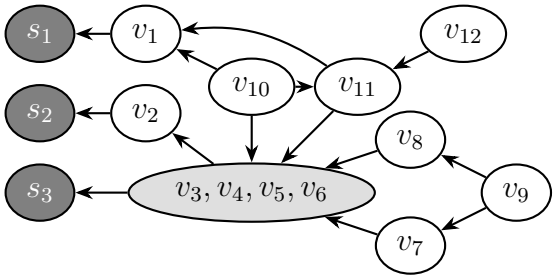


FIGURE 4.3. SCC-Condensation of the graph in Figure 4.1

Input: A sink graph $G = (V, S, E)$
Output: SCC-condensation of G
1 Compute the set of all SCCs in G
2 Condense G based on the equivalence classes defined by the SCCs

Algorithm 5: SCC-Condensation Algorithm (Operator S)

Each SCC defines an equivalence class under SCC-equivalence. To conduct SCC-condensation, we first compute all the SCCs and then contract each SCC into a super-vertex. The pseudocode is shown in Algorithm 5. It is well-known that the set of all SCCs in a directed graph can be

computed in linear time, e.g., by Tarjan's algorithm [75]. Consequently, SCC-condensation can be conducted in linear time. In the following, we call Algorithm 5 the condensation operator S.

4.3.1.2 DOM-Condensation

Our second condensation is based on the concept of dominance. In order to define dominance on a sink graph $G = (V, S, E)$, we need to introduce an exit vertex $\perp$ to G, and add an edge from every sink vertex $s \in S$ to $\perp$, e.g., see Figure 4.4. In the following, we assume that such an exit vertex $\perp$ always exists in G. Then, vertex $u \in V$ is said to *dominate* vertex $v \in V$ (or v is dominated by u), if every path from v to $\perp$ goes through u.

DEFINITION 6. *Given a sink graph $G = (V, S, E)$, two vertices $u, v \in V$ are said to be* DOM-***equivalent****, denoted $u \simeq_{\text{DOM}} v$, if either u dominates v or v dominates u in G.*

The relation induced by DOM-equivalence directly is not transitive. For example, it is possible that both u and w are dominated by another vertex v, but there is no dominance between u and w. We manually enforce transitivity, i.e., we also add (u, w) to the relation $\simeq_{\text{DOM}}$ if this happens. We call the resulting relation $\simeq_{\text{DOM}}$ the DOM-equivalence relation which is reflexive, symmetric, and transitive. It can be verified that $\simeq_{\text{DOM}} \subseteq \simeq_r$. Thus, the condensation induced the DOM-equivalence relation, called DOM-condensation, is sink-reachability preserving. For the sink graph in Figure 4.1, v_{11} dominates v_{12}, and v_3 dominates $\{v_5, v_6, v_7, v_8, v_9\}$; its DOM-condensation is shown in Figure 4.4. It is known in [40, 10] that the dominance relationship

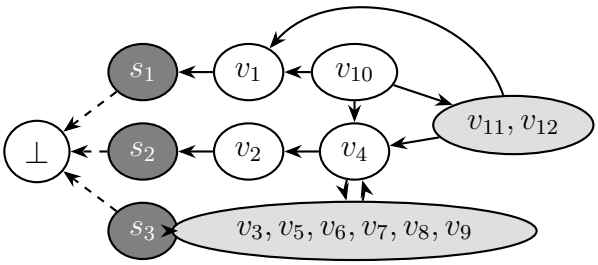


FIGURE 4.4. DOM-Condensation of the graph in Figure 4.1

among all vertices in a directed graph can be compactly represented by a dominator tree rooted at $\perp$, and the dominator tree can be constructed in linear time. Consequential, we can obtain the DOM-equivalence classes from the dominator tree in linear time. However, the linear-time dominator tree construction algorithms [10] are complicated. In this paper, we propose a simpler and practical algorithm to conduct DOM-condensation by

- directly obtaining the DOM-equivalence classes without constructing the dominator tree, and
- assuming that the sink graph is **acyclic** (which will be made possible in Section 4.3.2).

Input: An acyclic sink graph $G = (V, S, E)$
Output: DOM-condensation of G

```
Initialize a disjoint-set data structure 𝒟 for V
Compute a topological ordering of V
for vertex u ∈ V in reverse topological ordering do
    if N⁺(u) ∩ S = ∅ and N⁺(u) belong to the same set in 𝒟 then
        Union u and N⁺(u) in 𝒟
    end
end
Condense G based on the equivalence classes defined by 𝒟
```

Algorithm 6: DOM-Condensation Algorithm (Operator D)

The proof of the correctness of this algorithm and its time complexity as $O(|E|)$ is given in [19].

4.3.1.3 IMOD-Condensation and CMOD-Condensation

Our next two condensations are based on the concept of module [48, 23], which we adapt to sink graphs as follows. Given a sink graph $G = (V, S, E)$, a vertex subset $M \subseteq V$ is a *module* if all vertices of M have the same external out-neighbors, i.e., $N^+(u) \backslash M = N^+(v) \backslash M, \forall u, v \in M$. It is easy to see that all vertices in the same module are equivalent in the kernel equivalence relation $\simeq_r$. However, there are two challenges to compute modules.

- The existing studies on modular decomposition require all vertices in a module to have *the same external in-neighbors* as well as the same external out-neighbors, while we only require the same external out-neighbors.
- The existing algorithms designed for modular decomposition on directed graphs are of theoretical interest only, where no implementation exists.

Thus, we define the following two special cases of module that we can compute efficiently.

DEFINITION 7. *Given a sink graph $G = (V, S, E)$, two vertices $u, v \in V$ are said to be* IMOD-***equivalent****, denoted $u \simeq_{\text{IMOD}} v$, if $N^+(u) = N^+(v)$.*

DEFINITION 8. *Given a sink graph $G = (V, S, E)$, two vertices $u, v \in V$ are said to be* CMOD-***equivalent****, denoted $u \simeq_{\text{CMOD}} v$, if $(u, v) \in E$ and $N^+(u) \backslash \{v\} \subseteq N^+(v)$.*

The relation induced by IMOD-equivalence is reflexive, symmetric, and transitive. For CMOD-equivalence, we additionally enforce reflexivity, symmetry, and transitivity, similar to Section 4.3.1.2, to obtain a CMOD-equivalence relation. It is easy to see that $\simeq_{\text{IMOD}} \subseteq \simeq_r$ and $\simeq_{\text{CMOD}} \subseteq \simeq_r$. Thus, the condensation induced by the IMOD-equivalence relation, called IMOD-condensation, and the condensation induced by the CMOD-equivalence relation, called CMOD-condensation, are both sink-reachability preserving. For the sink graph in Figure 4.1, its IMOD-condensation is shown in Figure 4.5 and its CMOD-condensation is shown in Figure 4.6. Note that, alternatively we can change the condition in Definition 8 to be "$(u, v) \in E$ and $N^+(u) \backslash \{v\} = N^+(v)$". However, it can be verified that Definition 8 defines a larger equivalence relation and thus is better for condensation.

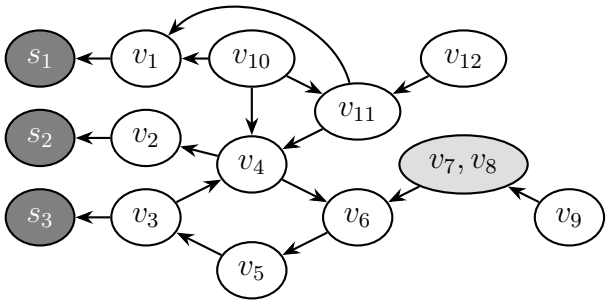


FIGURE 4.5. IMOD-condensation of the graph in Figure 4.1

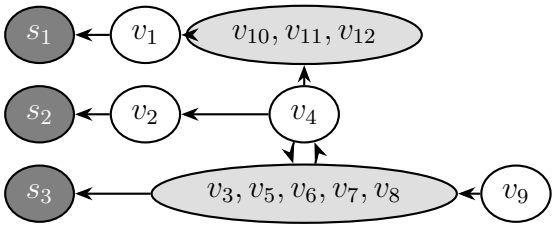


FIGURE 4.6. CMOD-condensation of the graph in Figure 4.1

Input: A sink graph $G = (V, S, E)$
Output: IMOD-condensation of G

1 Initialize a partitioning $\mathcal{P} = \{V\}$
2 **for** *vertex* $u \in V \cup S$ **do**
3 | Refine $\mathcal{P}$ based on $N^-(u)$ (i.e., each partition $P \in \mathcal{P}$ is split into $P \cap N^-(u)$ and $P \backslash N^-(u)$)
4 **end**
5 Condense G based on the equivalence classes defined by $\mathcal{P}$

Algorithm 7: IMOD-Condensation Algorithm (Operator $\mathsf{M_i}$)

The pseudocode of conducting IMOD-condensation is shown in Algorithm 7, which is self-explanatory. We call Algorithm 7 the condensation operator $\mathsf{M_i}$.

The proof of the correctness of this algorithm and its time complexity as $O(|E|)$ is given in [19].

Input: An acyclic sink graph $G = (V, S, E)$
Output: CMOD-condensation of G

1 Initialize a disjoint-set data structure $\mathcal{D}$ for V
2 Assign a topological number $t(u)$ to each vertex $u \in V \cup S$, and obtain a topological ordering of $V \cup S$
3 **for** *vertex* $u \in V$ **do**
4 | $t^{min}(u) \leftarrow \min_{v \in N^+(u)} t(v)$
5 **end**
6 **for** *vertex* $v \in V$ *in reverse topological ordering* **do**
7 | Mark v and $N^+(v)$
8 | **for** *in-neighbor* $u \in N^-(v)$ *of* v **do**
9 | | **if** $t^{min}(u) = t(v)$ ***and*** $d^+(u) - 1 \leq d^+(v)$ **then**
10 | | | **if** *all out-neighbors of* u *are marked* **then**
11 | | | | Union u and v in $\mathcal{D}$
12 | | | **end**
13 | | **end**
14 | **end**
15 | Unmark v and $N^+(v)$
16 **end**
17 Condense G based on the equivalence classes defined by $\mathcal{D}$

Algorithm 8: CMOD-Condensation Algorithm (Operator $\mathsf{M_c}$)

The pseudocode of conducting CMOD-condensation is shown in Algorithm 8, like Algorithm 6 it assumes that the sink graph is acyclic. The algorithm processes vertices in reverse topological ordering (Line 6). When processing vertex v, it checks for every in-neighbor $u \in N^-(v)$ whether $N^+(u) \backslash \{v\} \subseteq N^+(v)$: if the condition holds (Line 10), then u and v are CMOD-equivalent (see Definition 8). Here, for time efficiency consideration, we first conduct a constant-time filtering at Line 9: $t^{min}(u) = t(v)$ and $d^+(u) - 1 \leq d^+(v)$ which are necessary conditions for $N^+(u) \backslash \{v\} \subseteq N^+(v)$ to hold. We call Algorithm 8 the condensation operator $\mathsf{M_c}$.

The proof of the correctness of this algorithm and its time complexity as $O(|E|)$ is given in [19].

4.3.2 Composing Condensation Operators

In this subsection, we investigate how to compose the four condensation operators $\{\mathsf{S}, \mathsf{D}, \mathsf{M_i}, \mathsf{M_c}\}$ proposed in Section 4.3.1. Note that, as each condensation operator maps sink graphs to sink graphs, condensation operators can be composed.

DEFINITION 9. *Given two condensation operators c_1 and c_2, we define $c_1 \circ c_2(G)$, for any sink graph G, as $c_2(c_1(G))$ where $c_1(G)$ denotes the result of condensation of G by c_1.*

By definition, vertices of $c_1 \circ c_2(G)$ are nested sets of vertices from G, i.e., classes of classes. It is practical to flatten this structure by recursively aggregating the elements of the nested classes. Let $\simeq_1 \subseteq V \times V$ and $\simeq_2 \subseteq V_{\simeq_1} \times V_{\simeq_1}$ be the underlying equivalence relations of operator c_1 on G and operator c_2 on $c_1(G)$, respectively. We define the equivalence relation $\simeq_1 \simeq_2 \subseteq V \times V$ to be: $u \simeq_1 \simeq_2 v$ if and only if $[u]_{\simeq_1} \simeq_2 [v]_{\simeq_1}$. Then, the result of flattening $c_1 \circ c_2(G)$ is the same as the condensation of G induced by $\simeq_1 \simeq_2$. That is, *a composition of condensation operators also has an underlying equivalence relation*. In the following, we consider only flat condensation, and use $c_1 \circ c_2$ to denote the flat condensation of composing c_1 and c_2. We refer to a composition sequence of any positive number of condensation operators simply as a *condensation sequence*, and use $\mathbb{C}$ to denote a condensation sequence. Given any two condensation sequences $\mathbb{C}_1$ and $\mathbb{C}_2$, we say $\mathbb{C}_1(G)$ is the same as $\mathbb{C}_2(G)$ for a sink graph G, denoted $\mathbb{C}_1(G) = \mathbb{C}_2(G)$, if the underlying equivalence relation of $\mathbb{C}_1(G)$ is the same as that of $\mathbb{C}_2(G)$. We say $\mathbb{C}_1$ is the same as $\mathbb{C}_2$, denoted $\mathbb{C}_1 = \mathbb{C}_2$, if $\mathbb{C}_1(G) = \mathbb{C}_2(G)$ holds for every sink graph G.

A condensation is said to be maximal when applying any operator on the graph has no effect, i.e. no operator can further condense the graph. More formally:

DEFINITION 10. *Given a sink graph G and a set $\mathcal{C}$ of distinct condensation operators, a condensation sequence $\mathbb{C} \equiv c_1 \circ \cdots \circ c_n$, where $c_i \in \mathcal{C}$ for $1 \leq i \leq n$, is said to be **maximal** if $\mathbb{C} \circ c(G) = \mathbb{C}(G), \forall c \in \mathcal{C}$.*

4.3.3 Properties of Operator Composition

We now present some useful properties of this composition of operators that bring certain practical advantages to performance. The first property is that repeated operator application, regardless of order, will reduce the graph to the same fixpoint. That is, given any sink graph G, all maximal condensation sequences over $\{\mathsf{S}, \mathsf{D}, \mathsf{M_i}, \mathsf{M_c}\}$ reduce G to the same sink graph. The proof of this result is given in [19].

As each condensation operator runs in linear time, the running time of a reduction sequence grows linearly with its length (i.e., the number of condensation operators in the sequence). Thus, for any given sink graph G, it is ideal to have a shortest reduction sequence. We present maximal condensation sequences that are at most 3 times longer than the shortest condensation sequence, for any given sink graph G. Given any acyclic sink graph G, let C be the composition of any permutation of $\mathsf{D}, \mathsf{M_i}, \mathsf{M_c}$, and let C^* be the sequence of repeatedly applying C until convergence, then the length of C^* is at most three times the length of the shortest reduction sequence. Again, the proof of this result is omitted, and the reader is referred to [19].

The third property is that the reduced graph at the fixpoint is *not* the same as the kernel condensation $G_{\simeq_r}$. That is, there are sink graphs G such that all condensation sequences over condensation operators $\{\mathsf{S}, \mathsf{D}, \mathsf{M_i}, \mathsf{M_c}\}$ yield a reduced sink graph that is larger than $G_{\simeq_r}$.

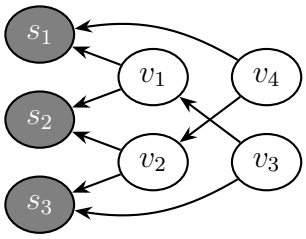


FIGURE 4.7. An example sink graph G where $G \neq G_{\simeq_r}$ but G cannot be reduced by $\{\mathsf{S}, \mathsf{D}, \mathsf{M_i}, \mathsf{M_c}\}$

We leave the study of efficiently computing the kernel condensation $G_{\simeq_r}$ to future work. Nevertheless, in practice maximal condensation sequences over $\{\mathsf{S}, \mathsf{D}, \mathsf{M_i}, \mathsf{M_c}\}$ compress sink graphs close to their kernel condensations, as we will show empirically in Chapter 5.

Our main results of composing condensation operators are summarized below.

Let $\mathcal{C}$ be $\{\mathsf{S}, \mathsf{D}, \mathsf{M_i}, \mathsf{M_c}\}$.

(1) Given any sink graph G, all maximal condensation sequences over $\mathcal{C}$ reduce G to the same sink graph.
(2) Given any sink graph G, the shortest maximal condensation sequence over $\mathcal{C}$ can be approximated within a factor of 3.
(3) There exist sink graphs G such that the reduced graph of G obtained by maximal condensation sequences over $\mathcal{C}$ is larger than the kernel condensation $G_{\simeq_r}$.

This properties afford our sink-reachability system key performance advantages, as we shall evidence in the experimental evaluation in the following chapter.

CHAPTER 5

Experimental Evaluation

5.1 Cloud-based Logic System

In this chapter, we conduct an experimental evaluation of the two systems involved in the hybrid approach. This first section regards the effectiveness of the distributed logic evaluation, presented in Section 3.1 and Section 3.2. We conduct our series of experiments to answer the following research questions regarding the costs, benefits, scalability, and performance of our approach.

(1) What is the communication cost and runtime overhead of the distribution?
(2) How effective is the stream-based, course-grained scalability, when used with a thread-based, fine-grained scalability?
(3) What is the difference in performance for a streaming logic evaluation in the cloud, against a local execution?
(4) What is the overhead of distributing heap objects (i.e. strings types), compared to values (i.e. integer types)?
(5) What is the effect of the program splitting technique, on both distributed and non-distributed programs?
(6) How does the distribution perform on real-world, and large-sized datasets?

5.1.1 Synthetic Weak-Scaling

Our first set of experiments evaluates the weak-scaling of our distributed logic evaluation against the baseline, non-distributed evaluation.

Weak-scaling, in which the available concurrency is increased with the data size, mitigates the bottleneck imposed by Amdahl's law [28] in the alternate strong-scaling method (where the data size only is changed, and the available concurrency fixed as constant). The approach mimics that used in the experimental evaluation of distributed SociaLite [64].

To answer RQ (1), we compare the distributed variant, which uses a *course-grained parallelism* across multiple actors, to the baseline, non-distributed variant, which uses a *fine-grained parallelism* across multiple threads. For each experiment, the *scaling factor* is doubled – this determines both the size of the input and the number of parallel computations, being threads in the non-distributed variant, actors in the distributed-variant.

The benchmark used for these experiments is the non-linear reachability program NR. For the non-distributed variant, NR is used in basic form, without any modification, as in Section 3.1, and reproduced below.

$$R_{xy} \text{ :- } E_{xy}.$$
$$R_{xz} \text{ :- } R_{xy}, R_{yz}.$$

For the distributed variant, NR is used with a number of splits equal to the scaling factor, as in Section 3.2, also reproduced below.

E_{xy} :- E^0_{xy}, $x \bmod N = 0$. ... E_{xy} :- E^i_{xy}, $x \bmod N = i$. ... E_{xy} :- E^{N-1}_{xy}, $a \bmod N = N-1$.

R^0_{xy} :- E^0_{xy}.
R^0_{xz} :- R^0_{xy}, R^0_{yz}. ...
R_{xy} :- R^0_{xy}. ...

R^i_{xy} :- E^i_{xy}.
R^i_{xz} :- R^i_{xy}, R^i_{yz}. ...
R_{xy} :- R^i_{xy}. ...

R^{N-1}_{xy} :- E^{N-1}_{xy}.
R^{N-1}_{xz} :- R^{N-1}_{xy}, R^{N-1}_{yz}.
R_{xy} :- R^{N-1}_{xy}.

R_{xy} :- E_{xy}.
R_{xz} :- R_{xy}, R_{yz}.

This means that running NR with multiple threads, in the non-distributed variant, partitions the E relation across n concurrent evaluations of the rule computing R, with n as the scaling factor). In the non-distributed variant, the rule for each split partition $E_1, E_2, \ldots E_n$ of E computes R_i at a separate node, before results are joined at another node R_n.

The scaling factor, being the number of threads or actors, begins at 1, and is doubled for each experiment until reaching a maximum of 32 threads or actors. The relationship between the input data and the scaling factor is rather more complicated. The input data with a scaling factor of 1 is a synthetic graph with 2^6 (64) vertices, with a scaling factor of 2, the graph has 2^7 (128) vertices, and so on – until with a scaling factor of 32, the graph has 2^{11} or 2048 vertices. The number of edges of the graph is always $\frac{|V|(|V|-1)}{2}$, and corresponds to the complete graph of a given size with a selection of half of the edges deleted. Each vertex has a out-degree between $\frac{|V|}{2} - 1$ and $\frac{|V|}{2}$. All synthetic graphs may contain cycles, do not have self-loops, and are directed. The synthetic datasets are presented below in Table 5.1.

TABLE 5.1. Synthetic datasets used in the distributed logic evaluation

Graph	n	$\|V\|$	$\|E\|$	min°	max°	avg°	$\frac{\|E\|}{n}$
synthetic-64	1	64	2016	31	32	31.5	2016
synthetic-128	2	128	8128	63	64	63.5	4064
synthetic-256	4	256	32640	127	128	127.5	8160
synthetic-512	8	512	130816	255	256	255.5	16352
synthetic-1024	16	1024	523776	511	512	511.5	32736
synthetic-2048	32	2048	2096128	1023	1024	1023.5	65504

The column for n is the scaling factor, $|V|$ and $|E|$ give the number of vertices and edges respectively, while min°, max°, and avg° give the minimum, maximum, and average number of (outbound) edges per vertex. $\frac{|E|}{n}$ gives the average number of edges communicated to, and computed upon, by each of the n split actors/threads.

The unique identifiers of the graph correspond to the index of the associated vertex label $G.L$, and as such, are successively increasing integers starting from 0. Recall that for each partition

E_i, with $0 \leq i < n$ where n is the number of partitions, we have the following rule.

$$E^i_{ab} \text{ :- } E_{ab}, E_{ab}, a \bmod n = i.$$

Hence, the application of the split rule induces a partition which is evenly distributed with regard to the unique identifiers (or, equivalently, label indices).

As each vertex has between $\frac{|V|}{2} - 1$ and $\frac{|V|}{2}$ edges, an even distribution of the unique identifiers corresponds to an even distribution of the number of edges across each split actor. In particular, each of the n splits is assigned $\frac{|E|}{n}$ edges, and as $|E| = \frac{|V|(|V|-1)}{2}$, each split actor gets a uniform $\frac{|V|(|V|-1)}{2} \times \frac{1}{n}$ edges each. As such, the total data volume, the data volume processed and communicated via each actor, and the number of actors involved in the concurrent evaluation,all increase proportionately to the scaling factor.

A plot of the results obtained is given below.

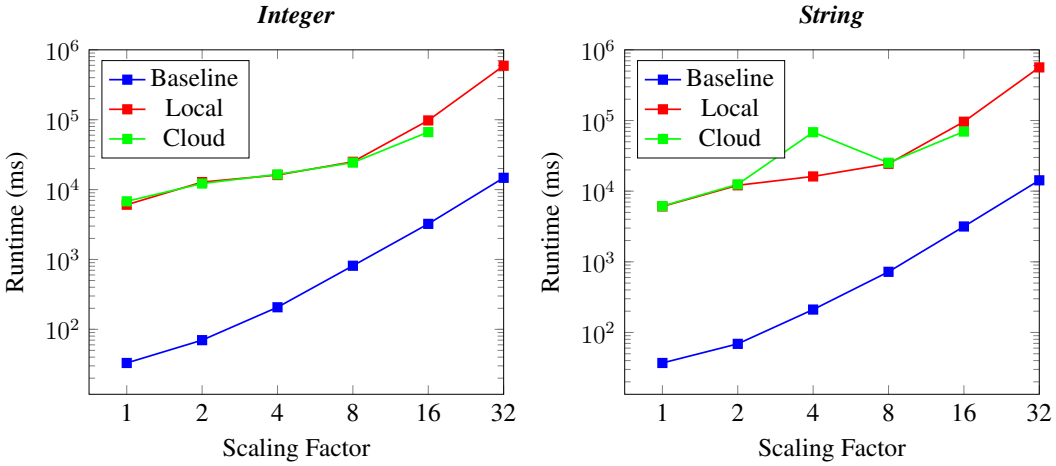


FIGURE 5.1. Comparison of runtimes of distributed and non-distributed evaluations in weak-scaling experiments.

The plot on the left shows the results for integer types, while the plot on the right shows results for string types. By including both plots, we may address RQ (4), by comparing the performance for simple data types, which are propagated directly between actors, to the performance for complex objects, which must be propagated via the use of shared dictionary passing as explained in Subsection 3.1.5. The x-axis of each plot shows the scaling factor, corresponding to the input graph and the number of threads/actors. The y-axis gives the runtime in milliseconds, with a logarithmic scale. The line titled "Baseline" is the non-distributed variant, in which standard logic evaluation is performed using unmodified Souffle, with a number of threads equal to the scaling factor. The "Local" line is the distributed Souffle on Kafka system, running each actor as a separate process on a single multicore machine, while the "Cloud" line is the distributed variant running in a cloud environment (Microsoft Azure), with each actor running in a virtual machine, each of which are distributed across multiple physical machines and cores thereof.

The results demonstrate that the non-distributed variant achieves within acceptable bounds of the best-case linear weak scaling. In answer to RQ (3), there was no meaningful difference in performance between the local and cloud deployment.

There is also no meaningful difference in performance between the integer and string typed executions. Hence, the overhead induced by strings is negligible, and that the distribution of complex objects scales for the benchmark. This additionally translates to the cloud domain, with similar performance in Azure and locally.

The only exception is a small spike for the cloud deployment for string typed data at a scaling factor of 4, likely due to the volatility of resources in the cloud environment. Additionally, in both graphs, both cloud and local deployment show nearly exactly the same runtime. The exception is that both integer and string typed cloud deployments reached a memory limit for the cloud environment at a scaling factor of 32, and so no result is shown for the cloud deployment at this scaling factor.

There is a large gap in raw performance (that is, runtime) between the non-distributed and distributed variant. This is to be expected. The distributed variant incurs a communication overhead associated with its distribution, and so for low scaling factors, the raw performance suffers. The runtime of the overhead is greater than any potential performance gains of the distribution in this case. As the runtime grows slower at higher scaling factors for the distributed variant, we can expect this overhead to be mitigated as the scale of the distribution increases. We strengthen this assertion based the analysis of the results in the plots that follow.

The results in Figure 5.2 are drawn from the same first round of experiments as Figure 5.1. However, they give a different measurement, showing the amount of time spent on message passing communication (shown in red) compared to the amount of time spent on computing the evaluation of the program (shown in blue). The communication time includes all time spent in initializing the connection to the distributed system (i.e. the client connection to the Kafka broker at each node), all time spent in message production and consumption subroutines, and any time spent polling the message queues to flush pending messages. The computation time includes all other operations that are not communication from the beginning of an actor's execution, and is measured as the communication time subtracted from the total execution time, local to each actor. The communication and computation times presented in the plots are the sums of all computation and communication times at each actor during its evaluation. The time in milliseconds is again shown on the y-axis with a log scale, and the scaling factor on the x-axis.

Plots are given for integer and string typed experiments, and the local execution with actors as processes on one machine (in the leftmost plots) is contrasted with the cloud execution where actors are distributed across virtual and physical machines of a cloud environment (in the rightmost plots). For the non-distributed baseline execution, the communication time is necessarily always zero, and the computation time simply the runtime of the execution as a whole; hence it is not necessary or meaningful to show results for the non-distributed variant in this set of plots.

We see that for both local and cloud deployments, with both integer and string types, computation time is initially, for the lower scaling factors, several orders of magnitude less than the

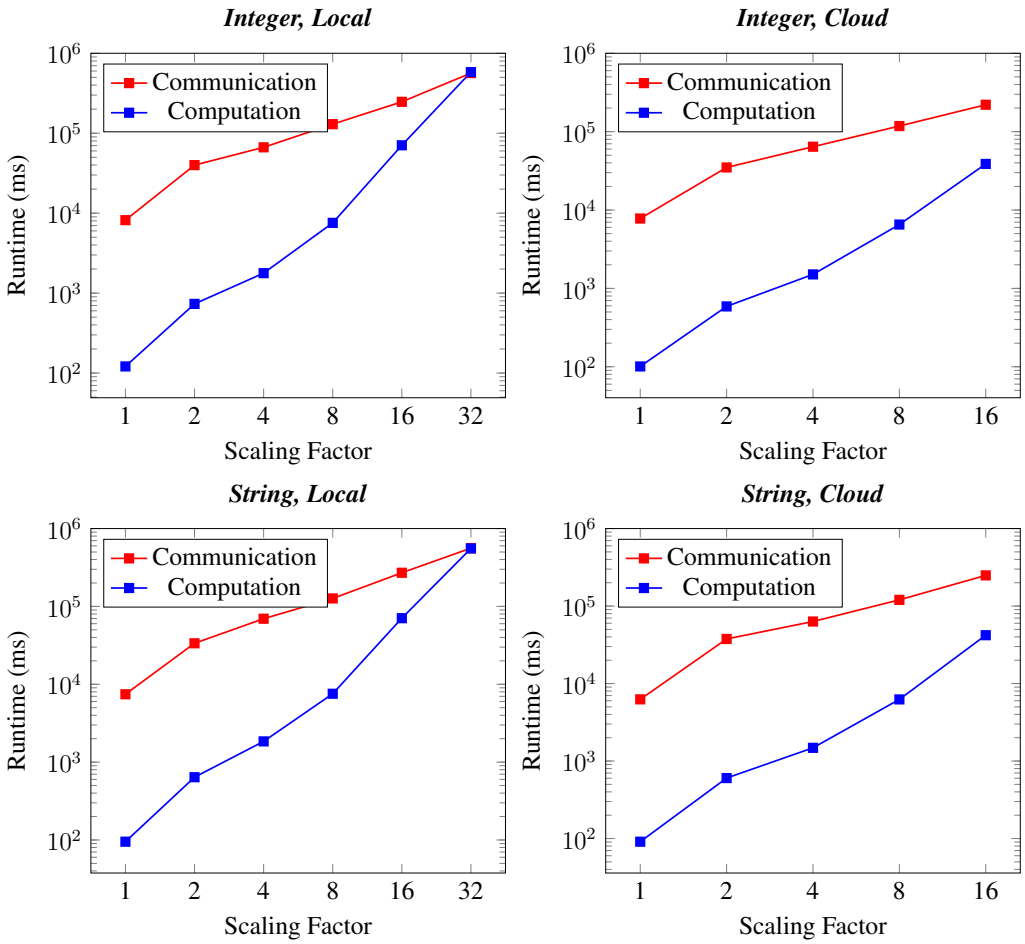


FIGURE 5.2. Comparison of communication and computation time in weak-scaling experiments.

communication time. This gap corresponds to the overhead introduced by the distribution, discussed prior. As the scaling factor is increased, the communication time grows at a slower rate than the computation time. For the local deployment, where the scaling factor reaches a maximum of 32, the lines for computation and communication time intersect. In the cloud deployment, where the maximum scaling factor reached was 16 due to a memory limit in the cloud environment, the lines follow the same trend as in the local deployment up to this scaling factor, and could reasonably be predicted to intersect at higher scaling factors.

The edges are partitioned uniformly between each of the n split across $E_1, E_2, \ldots, E_n$, and the number of edges in each partition increases with the scaling factor n. At each increase of scaling factor, there will necessarily be an increase in communication time, as more edges are being communicated to each of the n split actors, and more paths are discovered and communicated from each of the n split actors. Similarly, the computation time – which is dependent on the rule evaluation for the reachability relation R – will necessarily increase with the increase of the scaling factor n, as more edges are required to be processed at each

split node, and subsequently, more paths discovered. The plot captures a comparison between the proportionate increase in communication time against computation time relative to a given workload size. We expect the main overhead of our system to be the communication time, based on prior work. As the results show that computation time becomes the primary overhead at larger volumes of data, the scaling of the communication of the system is effective. For such compute heavy programs, we may increase the number of actors via program re-writes such as the splitting technique, which achieves an increased available parallelism with, as we have shown, acceptable weak scaling.

5.1.2 Fine-tuning of Course and Find Grained Parallelism

A substantial runtime performance gap is observed for the distributed evaluation as compared to the non-distributed evaluation, which our next set of experiments seeks to address. The baseline non-distributed variant, and both the cloud and local distributed variants, achieve an analogous concurrency in terms of how they perform the evaluation in parallel. The non-distributed variant partitions the workload for concurrent evaluation across cores of one node, while the distributed variant partitions this workload across actors having one core each. We can mitigate this performance deficit for the distributed evaluation, by combining the course-grained concurrency that distributes rule evaluation across actors, with the fine-grained concurrency that distributes rule evaluation across cores of the same node. This property of orthogonality for the course-grained scalability and the fine-grained scalability is advantageous, in that we may increase the number of cores utilized at each node of the split, without increasing the number of split actors, and thereby avoid adding to the communication overhead.

The next set of experiments involves a process of "fine-tuning", where the number of split actors and the number of cores used by each are varied independently, resulting in different performance. The results address 'rq12, showing the effectiveness of combining fine-grained and course-grained scalability, as well as 'rq15, by making use of the splitting technique to improve performance.

The plots again use the NR program as the benchmark, with both integer and string types, and compare the baseline execution (i.e. non-distributed evaluation) against the local execution (i.e. the distributed evaluation, across separate processes of the same node). The y-axis shows the runtime in milliseconds, scaled logarithmicly. The x-axis shows the ratio of splits used in the distribution, to the number of threads used by each stratum.

Recall that the split technique clones a program at the logic level, creating more rules, and hence strata, to evaluate relations. In the non-distributed evaluation, the strata of the program are evaluated in the total order of the topological sort of the SCC graph. This corresponds to the 'Baseline' line of the plot, with the given number of threads used in the evaluation of each stratum in sequence.

In the distributed evaluation, the strata are assigned to actors and evaluated concurrently at their own node. This corresponds to the "Local" line of the plot, where each actor is run as a separate process of a multicore machine, and thread count gives the number of threads used by each process. The dataset used is kept constant at all times during this experiment, and

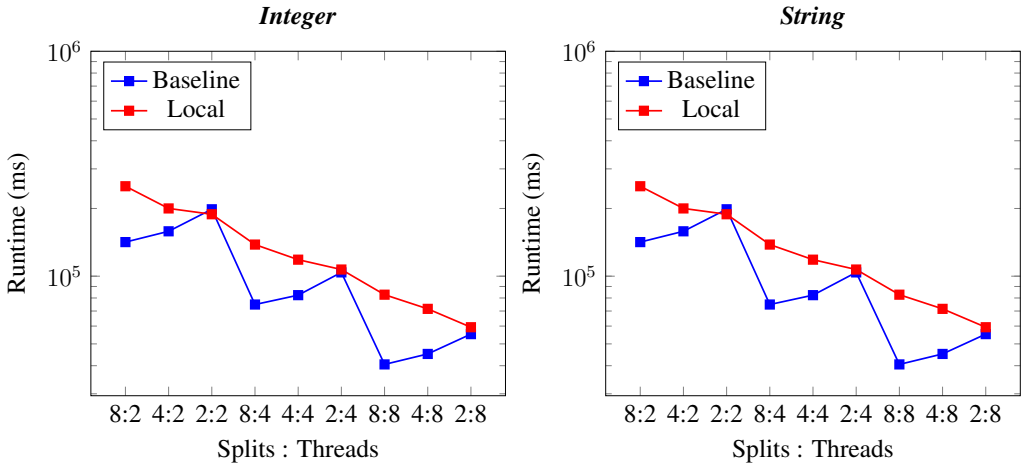


FIGURE 5.3. Comparison of runtimes of distributed and non-distributed evaluations in fine-tuning experiments.

is the maximum sized synthetic graph in the first set of experiments, with 2048 vertices and 2096128 edges.

There are several results demonstrated by the plot.

First, we see that the runtime of the baseline and the local evaluations are reduced by an increase in the number of threads used. This confirms the effectiveness of multithreading. The local evaluation evidences that multithreading is readily combinable as a means of adding fine-grained parallelism to a course-grained parallel evaluation. The baseline evaluation evidences multithreading as effective in its own right, confirming the fine-grained parallelism as efficient independent of other concurrency.

Second, for a fixed thread count, the runtime of the local evaluation increases with the increase of the split count. This evidences the disadvantage of the communication overhead on the distributed approach. While there is more available concurrency in the distributed variant, as multiple strata may be communicated in parallel by each of the split actors, the communication of data across these actors impacts the performance more than is compensated for by the reduction in computation time. We further demonstrate this in the comparison of communication and computation time for this round of experiments, presented in the plot below.

We see that, when the thread count is fixed, the computation time increases *slightly*, with a *decrease* in the number of splits. Conversely, with a fixed thread count, the communication time increases *substantially* more than the computation time, with an *increase* in the number of splits. This clearly demonstrates that the communication time overhead overshadows the benefit to the computation time obtained by increasing the number of splits.

This drawback is acceptable, as our the distributed variant's advantage lies in its deployment within cloud environments, and ability to be combined with other forms of concurrency. While in the distributed (local) variant, the runtime is higher at greater numbers of splits, we see

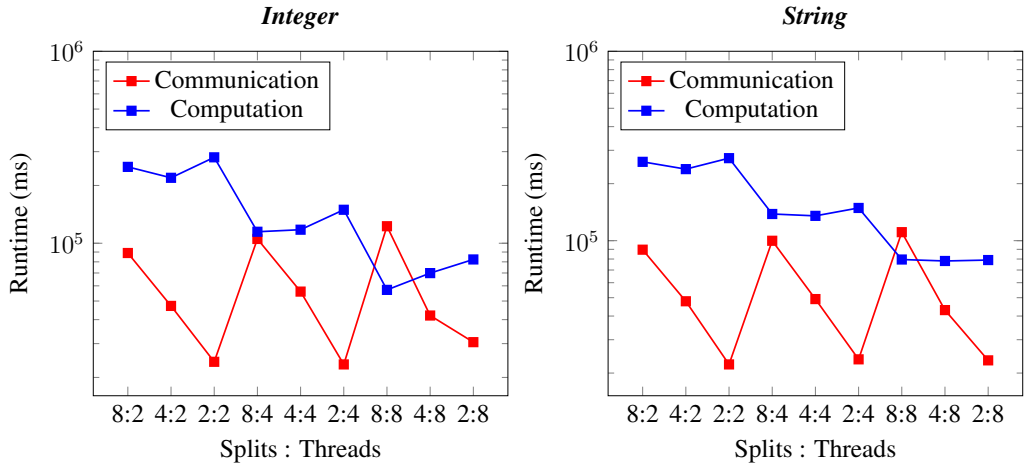


FIGURE 5.4. Comparison of communication and computation time in fine-tuning experiments.

that an increase in the available multithreading at each split can mitigate this communication overhead. Also, the raw performance of the distributed variant is always approximately greater than or equal to the non-distributed variant, the advantage of the distributed variant lies not in its total runtime benefit, but the infrastructure on which it may be deployed. The non-distributed variant must be deployed on a single multicore machine to reap the benefits of the fine-grained parallelism. The combination with the course-grained parallelism allows the distribution of the fine-grained parallelism across many multicore machines, but with a substantially lesser core count. This means that distributions may execute in cloud environments, across cheap commodity hardware, as opposed to singular massively multicore machines or specialized shared memory architectures, which are expensive and not readily available.

The third result is that for a fixed thread count, the runtime of the baseline evaluation decreases with the increase of the split count. That is, the runtime of the baseline is reduced by increasing the splits (for a fixed thread count). This is unexpected, as an increase in the split count corresponds to an increase in the number of strata that must be evaluated, and for the non-distributed evaluation, these must be evaluated in a sequential, blocking, total order. This implies that the split technique can be used to effectively speed up the evaluation of non-distributed programs. We conjecture that the split, by partitioning the relation over multiple rule evaluations, reduces the search space of the partitioned rule evaluations. As the relation's index, used for the lookup of tuples in the evaluation of a rule, is reduced in size, the search time, and hence rule evaluation itself, is reduced also. The results imply that this reduction in the index search time is significant enough to offset the increases in runtime resulting from the sequential blocking evaluation of the new strata introduced by the split.

The fourth result is that there is no significant performance deficit for the use of integer types against string types in this set of experiments. This re-enforces the effectiveness of the dictionary-based technique for passing heap object types, and further confirms the overhead mentioned in RQ (4) to be negligible.

5.1.3 Real-World Graph Datasets

To address RQ (6), the final set of experiments tests the performance of the system on three real world graphs, with between 4039 and 334863 vertices, and between 88234 and 925872 edges. We also compare these to the performance of the system on the largest synthetic graph, with 2047 vertices and 2096128 edge. The experiments here use the same NR benchmark program as previously, but this time with a fixed 2 splits and 16 threads, for both distributed and non-distributed evaluations. The real-world datasets used are given in Table 5.2.

TABLE 5.2. Real-world datasets used in the distributed logic evaluation

Graph	$\|V\|$	$\|E\|$	$\|zero^\circ\|$	min°	max°	avg°
amzn	334863	925872	68930	0	168	2.8
fb	4039	88234	376	0	1043	21.8
wiki	7115	103689	1005	0	893	14.6

Each of the real-world datasets is from the Stanford Network Analysis Project (SNAP). SNAP has real-world graphs of varying sizes and structures, drawn from a range of different problem domains, and used as common benchmarks in graph analysis [42]. In our experiments, amzn is the com-Amazon dataset, fb is the ego-Facebook dataset, and wiki is the wiki-Vote dataset, all from SNAP.

The amzn dataset, used in [81], is an undirected product co-purchasing network from Amazon.com. The graph has an undirected edge (u, v) if product u is frequently co-purchased with product v. The largest connected component is taken for the amzn dataset. For our experiments, we interpret each undirected edge in the dataset as a single directed edge.

The fb dataset, used in [47] is an undirected graph representing "friends-of" relationships for a subset of users of the Facebook social network. Again, each undirected edge becomes a single directed edge in the fb dataset used in our experiments.

The wiki dataset, used in [41], is a directed graph representing user votes in administrator elections on Wikipedia. Wikipedia administrators are contributors elected by the Wikipedia community to have certain elevated privileges over content. The dataset contains all Wikipedia voting data from the beginning of Wikipedia up to January 2008, where a directed edge (u, v) indicates that user u voted on user v.

Each of the real-world datasets is pre-processed before use in the experiments. Initially, graphs were transformed into edge lists, with one pair of vertex labels per directed edge in the list. Then, each vertex label is replaced by an index, starting at 0, and successively increasing by 1 for each newly discovered label. The edge lists with the new labels are then written to files in the ".facts" format read by Datalog programs, with one pair of vertex labels per line. The re-labeling of the vertices allows us to consider the labels themselves as either integer or string typed in the NR program. This method of using numerals as vertex labels is the same technique we have used previously to compare performance on executions involving integer types against those on strings in previous experiments.

The runtime results of this round of experiments is shown in the plot above. We observe that the runtime of the amzn and fb datasets is several orders of magnitude greater for the

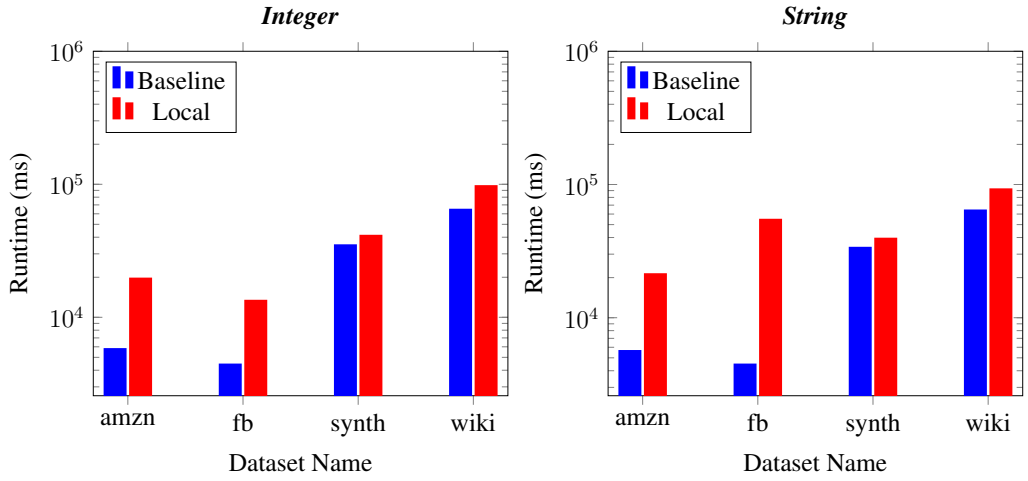


FIGURE 5.5. Comparison of runtimes of distributed and non-distributed evaluations in experiments on real-world datasets and largest synthetic dataset.

distributed evaluation (local) than the non-distributed evaluation (baseline). For the synth and wiki datasets, however, the performance is marginally better for the non-distributed than the distributed. Comparing the communication and computation time of the distributed variant gives explanation to these results.

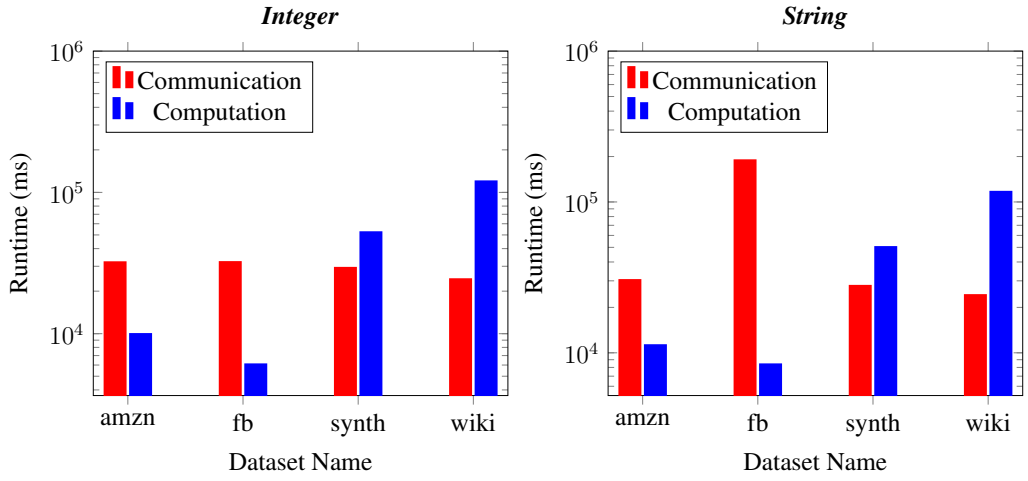


FIGURE 5.6. Comparison of communication and computation time in experiments on real-world datasets and largest synthetic dataset.

We see that the graphs which take the most runtime (amzn, fb) in our experiments correspond to a distinctly higher communication load than computation load. For the graphs where the runtime of the non-distributed variant is closer to the distributed variant (synth, wiki), the computation time is greater than the communication.

The key insight of this result is that graphs which induce a higher communication than computation load do not scale effectively with the distributed evaluation. Moreover, choices

of dataset exhibit vastly different performance characteristics even when the distribution (in terms of threads and splits) is kept fixed, and the graphs themselves are restricted to a range of sizes (in terms of vertex and edge count), and graph densities (in terms of average neighbor count of the vertices).

The real-world graphs are chosen as to represent a variety of alternate structures within a restricted range of sizes. The best performance dataset for the distributed evaluation was the synth dataset. The thread and split count for the distribution was determined by the results of the earlier fine-tuning experiments on this same synth graph. As the fine-tuning experiments demonstrated how the alternative choice of splits and threads can be used to optimize performance, the result of synth being the best for this distribution is expected.

The amzn and fb graphs do not show acceptable performance for the distributed evaluation against the baseline. This shows that the fine-tuning does not translate generally to performance on different graphs, even when the sizes are similar, and the benchmark program used is the same. The wiki graph, however, does show acceptable performance under the distribution resulting from the fine-tuning the the synth graph. This means that the fine-tuning approach does translate between different datasets, in some cases. Hence, the distribution technique has been shown to be effective for a particular selection of program, dataset, and distribution schema. Beyond this, the result is somewhat inconclusive – further work is required to more precisely investigate the relationship between program, dataset, and distribution.

The raw performance, in terms of runtime, is always greater than or equal for the distributed variant against the non-distributed variant. We have, thus far, taken this as an acceptable performance deficit, as the distribution scales across actors *and* cores thereof (as opposed to cores only), and offers many other advantages due to its feature-rich cloud deployment. We have also shown that alternative combinations of fine-grained and course-grained parallelism, choice of program, and choice of dataset can improve performance dramatically – developing better models for how these factors interact would likely afford further increases in efficiency.

This concludes the experimental evaluation on the first part of our hybrid approach. The results show that the actor-based distribution scales for use within cloud environments, where combining course-grained, stream-based parallelism with fine-grained, thread-based parallelism enables improvements. We now move on to the second experimental evaluation for our hybrid approach. We target only raw performance as our main concern, on a specific, high-impact problem from the logic programming domain. We trade the genericity of the streaming, actor based distribution, which targets all possible bottom-up logic programs, for single-purpose, optimized algorithms in hand-written C++. These algorithms may be included within logic programs via user-defined functions, in combination with, and orthogonal to, both the course-grained concurrency introduced in this work, and the existing fine-grained concurrency.

5.2 Extra-logical Sink-Reachability Framework

In this second section of our experimental evaluation, we investigate the performance of the sink-reachability preserving graph compression framework.

The first component of the hybrid approach, the distributed logic evaluation, is generic, applicable to any bottom-up logic program, and intended for practical, cloud based distribution, as opposed to performance. This second component of the hybrid approach, the graph reduction framework, is designed specifically for the problem of sink-reachability. The implementation is hand-written in C++, does not admit a distributed evaluation, and is intended for one purpose only – computing sink reachability with state of the art performance. The hybrid aspect of our technique lies in combining these two approaches, to bring novel scalability to feature-rich logic programs in cloud environments, together with performance on par with native code implementations for specific, high-impact problems. We conduct extensive experiments with large-scale graphs from static program analysis, social networks and web graphs from SNAP. Experiments on large real-world sink graphs demonstrate the efficiency and effectiveness of our composition framework. We can show compression rate of up to 99.74% for vertices and a compression rate of up to 99.46% for edges. The empirical results presented in this section answer the following research questions of our sink-reachability preserving graph compression framework.

(1) How effective are the individual linear time sink-reachability preserving compression operators (S, D, M_i, and M_c)?
(2) How effective is the composition of linear time sink-reachability preserving compression operators in sequence?
(3) How close is the length of the compression sequence achieved in practice to the optimal compression sequence?
(4) Which sequence of linear-time sink-reachability preserving compression operators is most effective in practice?
(5) Is the compression achieved by composition of linear-time sink-reachability preserving compression operators effective for computing the reachability index of sink-graphs?

We maintain a strong focus for our research questions on the linear-time sink-reachability preserving compression operators. We do not include a comparison to logic programs, either distributed or otherwise. This is because when tested with simple sink-reachability logic programs,given the same compute resources as we afford to our framework in the following experiments, preliminary experiments on even the smallest sized datasets used here reached a timeout or memory limit. The graph processing framework, put simply, is in a different league to what can be achieved by a standard logic evaluation, even using a highly optimized logic engine.

The framework is designed specifically with a focus on sink-reachability queries. As such, we utilize all other non-sink-reachability related operators of the framework for comparison purposes only, in order to determine whether the sink-reachability preserving compression

approach offers particular advantage for computing sink-reachability queries over that of existing techniques.

5.2.0.1 Datasets

We present the datasets used in this set of experiments, and statistics thereof.

Note that the sizes of datasets are much greater than in our earlier section on the experimental evaluation of the distributed logic system (see Section 5.2). As the problem of efficiently computing a points-to analysis is the motivating application of sink-reachability preserving compression, the majority of datasets – 15 in total – are selected from this domain. 13 of these are mid-sized Java programs from the well known Dacapo benchmarks [7].

The remaining 2 are *openjdk8*, an implementation of the Java standard library, and *jenkins*, which is a popular open source CI server software.

The sink-graphs themselves are extracted from the Java programs using the DOOP static-analysis framework [7]. DOOP allows us to convert the assignments and call statements into graphs over variables and the heap locations of objects they point to, in a ".facts" format suitable for logic programs. The variables become non-sink vertices, while heap locations become sinks.

The 15 graphs with the label suffix of "-s" in the table (meaning small, or sparse) are unsound, in the sense that fields are not modeled, and methods are not devirtualized – that is, they do not represent all possible runtime behaviors of the corresponding program. In order to bring soundness to the model, additional edges are added in the manner of [71], such that flow through fields for matching load-store records is captured. The resultant 15 sound graphs have the label suffix of "-l" in the table (meaning large, or dense).

Additionally, four datasets are used from SNAP (see Leskovec *et al.* [42]), though these are substantially larger in size than those SNAP datasets used for the experiments on the distributed logic system, and are drawn from different problem domains. The SNAP graphs are LiveJournal1, soc-Pokec, web-Berkstan, and web-Google – two are large social networks, the other two are large web graphs. LiveJournal1 (named soc-LiveJournal1 in the SNAP benchmark) is a directed social-network graph used in the work of [6] and [43]. The edges of the graph represent (directed) friends-of relationships for the popular online blog platform LiveJournal, with the nodes being users of the platform.

soc-Pokec dataset is also a directed social-network graph, used in the work of [74]. As of 2012, Pokec was the most popular social network in Slovakia (even more so than Facebook), had been in existence for 10 years, and connected more than 1.6 million people. The soc-Pokec dataset represents friends-of relationships on this social network.

The web-BerkStan dataset is a directed web-graph, used in the work of [43]. The graph itself represents hyperlinks (edges) between pages (nodes) on a snapshot of the berkeley.edu and stanford.edu domains, from 2002.

TABLE 5.3. Statistics of sink graphs $G = (V, S, E)$, where $|V_r|$ represents the vertex count in the kernel condensation, $|V^c|$ and $|E^c|$ represent the vertex count and edge count in the core graph (i.e., after removing all vertices that cannot reach any sink vertices)

Graphs	$\|V\|$	$\|S\|$	$\|E\|$	DAG?	$\|V^c\|$	$\|E^c\|$	$\|V_r\|$
avrora-s	562615	68971	708614	no	248066	483220	10340
batik-s	684010	78921	864264	no	290420	571600	12560
eclipse-s	324226	41516	414638	no	146513	289207	6183
fop-s	765724	95327	989696	no	339797	674585	13524
h2-s	590845	70293	758421	no	259960	514197	10402
jenkins-s	3196349	404514	4103165	no	1398361	2750669	41580
jython-s	867730	88565	1155836	no	352899	675566	12356
luindex-s	317197	41454	401044	no	143613	282322	5996
lusearch-s	317510	41503	401416	no	143639	282425	5997
openjdk8-s	1573653	215902	1978755	no	694939	1346830	22324
pmd-s	568521	67999	725670	no	247054	482285	10258
sunflow-s	522667	64464	664208	no	230685	450004	9880
tomcat-s	303510	40426	390167	no	141100	277496	5938
tradebean-s	272083	36272	346088	no	125810	246038	5356
xalan-s	673681	99952	841457	no	311060	601623	10908
avrora-l	562615	68971	2725835	no	443283	2369919	11794
batik-l	684010	78921	3336522	no	533147	2910704	14339
eclipse-l	324226	41516	1421235	no	248581	1233657	7160
fop-l	765724	95327	4063315	no	603459	3573151	15449
h2-l	590845	70293	2865741	no	460667	2480215	11964
jenkins-l	3196349	404514	59821383	no	2466802	53310172	48198
jython-l	867730	88565	26716793	no	701498	25943050	13676
luindex-l	317197	41454	1401880	no	245739	1222298	7178
lusearch-l	317510	41503	1402425	no	245938	1222868	7180
openjdk8-l	1573653	215902	12387187	no	1257860	11035312	26755
pmd-l	568521	67999	3068722	no	441828	2657647	11726
sunflow-l	522667	64464	2482443	no	409761	2152612	11171
tomcat-l	303510	40426	1327010	no	233842	1152127	6739
tradebean-l	272083	36272	1139807	no	209582	993454	6224
xalan-l	673681	99952	2997836	no	520315	2593130	12530
LiveJournal1-8	893533	77699	1017575	yes	366304	522267	7390
soc-Pokec-8	299821	26071	377788	yes	112609	168423	2825
web-BerkStan-8	100654	8752	574228	yes	81157	553039	209
web-Google-8	342023	29741	508426	yes	156756	263057	3536

web-Google is a directed web-graph presented in the same paper as web-Berkstan (see [43]). The is also a snapshot of web pages as nodes, and hyperlinks thereof as edges, provided by Google as part of a programming context in 2002.

As the SNAP graphs are not sink-graphs, we must assign a subset of their vertices as sinks to compute the sink-reachability preserving compression. We do this by use of operator **A**, which contracts SCCs (via a call to the algorithm of the **S** operator), computes a topological order of the resulting DAG, and assigns the last $x\%$ as sink vertices (by use of parameters `-population=`x and `-sample=1.0`). The values of x are $\{2, 4, 6, 8, 10, 20, 40\}$, with $x = 8$ as the default (shown in Table 5.3 above with suffix `-8`).

The statistics of the datasets are summarized in Table 5.3. The columns $|V|, |S|, |E|$ represent the number of non-sink vertices, sink vertices, and edges respectively. As evidenced by the $DAG?$ column, the graphs extracted from programs contain cycles, and the SNAP datasets, which are transformed by the **S** operator implementing the SCC reduction, are acyclic. The $|V^c|$ and $|E^c|$ columns give the vertices and edges (respectively) in the "core graph" – that is the graph obtained by applying the **T** operator to remove all non-sink vertices that do not reach a sink (note that $|S| = |S^c|$ always). We also give the number $|V_r|$ of non-sink vertices in the kernel condensation achieved by application of the **V** operator. This closely approximates the best sink-reachability preserving graph compression that is possible, and is used in forthcoming results to assess the quality of alternative reduction methods.

5.2.1 Effectiveness of Sink-reachability Preserving Compression Operators

In our first set of experiments for this section, we assess the effectiveness of the linear-time sink-reachability preserving compression operators, both individually to answer RQ (1), and in composition to answer RQ (2).

Table 5.4 shows the results of this first round of experiments.

We evaluate the effectiveness of the operators independently based on the results in column 2 to 9 of Table 5.4. We show the percentage of vertices retained in the transformed graph resulting from an operator application against the core graph $G^c = (V^c, E^c)$. For example, column 2 shows $\frac{|V_S|}{|V_c|} \times 100$, which compares the number of vertices for the graph transformed by the SCC operator **S** against the core graph. In terms of CLI options, `-prefix=OCTS` is used for G_c, while `-prefix=OCT` for the other operators – that is, the core graph first has cycles and non-sinks that don't reach sinks removed, while the graphs for the other operators only have non-sinks that don't reach sinks removed prior to subsequent operator application. Note that the sequences $S \circ D$, $S \circ M_i$, and $S \circ M_c$ are used for operators D, M_i, M_c respectively, as they require an acyclic input graph (as results from application of operator S).

For the sink-reachability operators compared independently, we make several observations from Table 5.4. We see that S has little to no effect for most graphs, retaining a large percentage of vertices and edges as compared to the core graphs. The D and M_c operators display the best reduction in most cases, while operator M_i obtains a reasonable reduction, but performs worst than D or M_c, and so is somewhat of a middle ground. Importantly, D, M_i, and M_c are shown to complement each other – they achieve a different reduction effectiveness, and hence, a different reduction.

TABLE 5.4. Percentage (%) for vertex ($|V|$) and edge ($|E|$) counts of the compressed graphs to the corresponding core graphs. $|V_r|$ and $|E_r|$ represent the size of the kernel condensation. $|V_{fp}|$ and $|E_{fp}|$ represent the size of the fixpoint condensation by $\{\mathsf{S}, \mathsf{D}, \mathsf{M_i}, \mathsf{M_c}\}$. $|V_{ter}|$ and $|E_{ter}|$ represent the size of the graph obtained by the techniques in [87]. Note that for the vertex count, only non-sink vertices are taken into account.

Graphs	$\frac{\lvert V_{\mathsf{S}}\rvert}{\lvert V^c\rvert}$	$\frac{\lvert E_{\mathsf{S}}\rvert}{\lvert E^c\rvert}$	$\frac{\lvert V_{\mathsf{SoD}}\rvert}{\lvert V^c\rvert}$	$\frac{\lvert E_{\mathsf{SoD}}\rvert}{\lvert E^c\rvert}$	$\frac{\lvert V_{\mathsf{SoM_i}}\rvert}{\lvert V^c\rvert}$	$\frac{\lvert E_{\mathsf{SoM_i}}\rvert}{\lvert E^c\rvert}$	$\frac{\lvert V_{\mathsf{SoM_c}}\rvert}{\lvert V^c\rvert}$	$\frac{\lvert E_{\mathsf{SoM_c}}\rvert}{\lvert E^c\rvert}$	$\frac{\lvert V_r\rvert}{\lvert V^c\rvert}$	$\frac{\lvert E_r\rvert}{\lvert E^c\rvert}$	$\frac{\lvert V_{fp}\rvert}{\lvert V^c\rvert}$	$\frac{\lvert E_{fp}\rvert}{\lvert E^c\rvert}$	$\frac{\lvert V_{ter}\rvert}{\lvert V^c\rvert}$	$\frac{\lvert E_{ter}\rvert}{\lvert E^c\rvert}$
avrora-s	94.15	93.70	6.41	39.10	61.94	53.83	7.47	25.56	4.17	21.44	4.40	21.75	73.01	59.32
batik-s	94.64	94.01	6.91	39.74	61.92	53.82	7.96	26.40	4.32	21.79	4.56	22.09	73.59	59.63
eclipse-s	93.92	93.32	6.81	39.56	61.50	53.30	7.96	25.92	4.22	21.41	4.44	21.66	72.81	58.61
fop-s	94.04	93.67	6.15	39.86	60.75	53.60	7.17	26.42	3.98	22.54	4.16	22.78	71.55	58.89
h2-s	93.30	92.75	6.54	38.45	59.70	51.62	7.67	25.21	4.00	20.67	4.24	20.97	70.92	57.22
jenkins-s	93.68	93.17	4.91	37.16	55.95	48.83	5.71	22.93	2.97	19.13	3.11	19.31	66.80	53.53
jython-s	94.64	94.08	5.65	36.36	57.08	50.41	6.61	23.88	3.50	19.87	3.69	20.10	67.73	55.86
luindex-s	93.51	92.93	6.57	39.47	62.17	53.53	7.64	25.45	4.18	21.23	4.39	21.47	72.99	58.82
lusearch-s	93.50	92.92	6.56	39.49	62.17	53.52	7.63	25.44	4.18	21.24	4.39	21.48	72.98	58.81
openjdk8-s	95.09	94.61	5.33	38.30	58.26	49.73	6.27	22.94	3.21	19.00	3.37	19.20	68.24	53.86
pmd-s	93.86	93.36	6.52	38.96	61.40	53.40	7.58	25.52	4.15	21.28	4.39	21.58	72.59	58.94
sunflow-s	94.41	93.92	6.69	39.38	61.66	53.60	7.76	25.75	4.28	21.48	4.53	21.78	72.80	59.08
tomcat-s	93.76	93.17	6.69	39.47	61.67	53.29	7.78	25.56	4.21	21.24	4.44	21.50	72.87	58.76
tradebean-s	93.99	93.36	6.80	39.62	62.12	53.54	7.85	25.48	4.26	21.06	4.49	21.33	73.23	59.00
xalan-s	94.34	93.86	5.64	38.84	60.25	50.67	6.54	22.82	3.51	19.02	3.71	19.26	70.00	55.56
avrora-l	87.54	66.31	12.76	34.91	50.82	19.99	11.75	15.77	2.66	6.28	3.07	6.88	66.29	21.90
batik-l	87.15	64.90	13.86	35.58	49.88	20.00	12.81	16.71	2.69	6.35	3.13	7.03	65.46	21.10
eclipse-l	87.11	69.13	13.66	37.88	51.00	22.08	12.68	17.93	2.88	7.13	3.34	7.77	66.15	25.83
fop-l	86.87	62.92	13.07	34.42	49.72	19.13	12.14	16.10	2.56	6.38	2.98	6.96	64.89	20.24
h2-l	86.64	66.09	13.34	34.58	48.97	19.78	12.37	16.32	2.60	6.34	3.03	6.93	64.34	21.63
jenkins-l	85.12	50.55	11.11	31.39	46.27	6.89	10.00	6.77	1.95	1.81	2.25	2.11	60.63	5.45
jython-l	74.80	10.56	10.56	5.84	40.68	2.47	9.55	2.17	1.95	0.76	2.25	0.84	53.32	2.72
luindex-l	87.45	70.72	14.18	39.31	51.55	22.32	13.16	18.84	2.92	7.28	3.40	7.98	66.72	26.56
lusearch-l	87.46	70.74	14.19	39.32	51.56	22.33	13.17	18.86	2.92	7.28	3.40	7.98	66.74	26.59
openjdk8-l	87.51	56.15	12.06	30.81	48.23	13.04	11.47	11.56	2.13	3.90	2.47	4.38	63.08	13.11
pmd-l	85.83	63.63	13.31	34.81	49.50	18.12	12.09	15.03	2.65	5.80	3.06	6.34	64.57	19.84
sunflow-l	87.04	65.68	13.28	35.32	50.07	20.43	12.20	16.74	2.73	6.57	3.16	7.18	65.20	22.37
tomcat-l	86.68	67.35	13.51	36.97	51.34	21.89	12.04	16.74	2.88	7.10	3.33	7.69	66.08	24.20
tradebean-l	87.37	70.46	14.17	38.79	51.68	22.89	13.11	18.60	2.97	7.26	3.45	7.94	66.69	27.23
xalan-l	87.66	66.83	12.38	34.90	50.19	20.95	11.33	16.71	2.41	6.45	2.81	7.06	64.20	22.56
LiveJournal1-8	100.00	100.00	2.39	30.59	5.14	18.49	3.12	15.95	2.02	15.13	2.02	15.13	7.16	18.68
soc-Pokec-8	100.00	100.00	3.35	33.36	6.09	21.29	4.68	18.31	2.51	16.73	2.52	16.74	8.25	20.46
web-BerkStan-8	100.00	100.00	2.42	2.99	6.54	3.10	25.97	7.33	0.26	0.54	0.26	0.54	11.77	3.14
web-Google-8	100.00	100.00	3.34	21.96	15.36	25.93	16.55	19.88	2.26	9.13	2.27	9.15	25.31	25.70

Columns 10 to 15 of Table 5.4 contain results by which we evaluate the effectiveness of the *composition* of linear-time sink-reachability preserving graph compression operators. The columns for $|V_r|$ and $|E_r|$ give the compression of the approximate vertex kernel obtained by operator **V** against the core graph, providing a baseline of an (approximate) best possible compression. The columns for $|V_{fp}|$ and $|E_{fp}|$ give the maximal compression achieved by repeated application of the linear-time sink-reachability preserving compression operators. This is the compression obtained by repeated application of S, D, M_i, M_v until the graph reaches a fixpoint where the application of any of these operators does not further reduce the graph – a unique fixpoint is guaranteed regardless of operator application order by the fixpoint

theorem in [19]. The columns for $|V_{ter}|$ and $|E_{ter}$ are the current state of the art reachability-preserving compression as proposed in [19]. This utilize the **B** operator, for bottom-up transitive reduction), followed by the **L** operator, for reduction via linear equivalence. These operators preserve general reachability, not sink-reachability specifically, and thus require more information to capture their reachability relation.

From Table 5.4, we observe that the fixpoint compression obtained via use of the sink-reachability preserving compression operators is close to the kernel condensation. However, in many cases, the fixpoint is slightly larger than the kernel, as there are kernel graphs that cannot be achieved by application of S, D, M_c, M_i alone [19]. Conversely, the graphs obtained by the **B** and **L** operators as per the techniques in [87] are all substantially larger than those of the sink-reachability preserving operators. This demonstrates the effectiveness of the composition of the sink-reachability preserving compression operators in reducing the graph.

5.2.1.1 Length of Compression Sequences

Our next set of experiments compares the length of the sequences formed by composing the sink-reachability preserving compression operators S, D, M_i, M_c, in order to answer RQ (3). As all such operators run in linear time of $O(|V| + |E|)$, the length k of the compression sequence determines the (asymptotic) runtime as $O(k(|V| + |E|))$. Thus, the shorter the compression sequence, the shorter the runtime. Also too, by the fixpoint theorem in [19], any compression sequence composed of all three sink-reachability compression sequence results in the same graph as any other such compression sequence involving the same operators, when operators are repeatedly applied. The compression sequence lengths for this experiment give the number of operators used up to this fixpoint, at which point the application of any other sink-reachability preserving compression operator has no effect on the graph.

We compare a selection of alternative compression sequences in Table 5.5 below on all datasets. We use the results to compare the compression sequence lengths against the shortest possible compression sequence, and evaluate the best choice of compression sequence to use in practice.

Table 5.5 shows the compression sequence lengths for each dataset, with columns labeled with the regex of the compression sequence used.

The optimal compression sequence length $|C_{opt}|$ gives the lowest possible number of sink-reachability preserving operator applications resulting in a fixpoint. We do this by passing the `-c`/`-count` option with the argument "`max`" in the invocation of the CLI. As this optimal sequence is found by a pruned complete search over all possible reduction sequences, the runtime is exponential in the worst case – hence finding the shortest sequence is not always feasible in practice.

The shortest compression sequence apart from C_{opt} is highlighted in **bold font** for each row and group of columns, where multiple compression sequences often result in the same length, and so are equally minimal. We observe that no compression sequence is greater than 3 times the length of C_{opt}, confirming the result in [19] that the maximum possible fixpoint

TABLE 5.5. Lengths of compression sequences ($\mathbb{C}_{opt}$ is the shortest compression sequence, $\mathbb{C}_{(\mathsf{DM_iM_c})^*}$ is the compression sequence $\mathsf{S} \circ (\mathsf{D} \circ \mathsf{M_i} \circ \mathsf{M_c})^*$, $\mathbb{C}_{\mathsf{D}(\mathsf{M_iM_c})^*}$ is the compression sequence $\mathsf{S} \circ \mathsf{D} \circ (\mathsf{M_i} \circ \mathsf{M_c})^*$, and $\mathbb{C}_{(\mathsf{M_iM_c})^*}$ is the compression sequence $\mathsf{S} \circ (\mathsf{M_i} \circ \mathsf{M_c})^*$)

Graphs	$\lvert\mathbb{C}_{opt}\rvert$	$\lvert\mathbb{C}_{(\mathsf{DM_iM_c})^*}\rvert$	$\lvert\mathbb{C}_{(\mathsf{DM_cM_i})^*}\rvert$	$\lvert\mathbb{C}_{(\mathsf{M_iDM_c})^*}\rvert$	$\lvert\mathbb{C}_{(\mathsf{M_iM_cD})^*}\rvert$	$\lvert\mathbb{C}_{(\mathsf{M_cDM_i})^*}\rvert$	$\lvert\mathbb{C}_{(\mathsf{M_cM_iD})^*}\rvert$	$\lvert\mathbb{C}_{\mathsf{D}(\mathsf{M_iM_c})^*}\rvert$	$\lvert\mathbb{C}_{(\mathsf{M_iM_c})^*}\rvert$
avrora-s	11	**16**	19	**16**	**16**	19	19	**13**	**13**
batik-s	12	**16**	19	**16**	**16**	19	19	**13**	**13**
eclipse-s	11	**16**	19	**16**	**16**	19	19	**13**	**13**
fop-s	10	**16**	19	**16**	**16**	19	19	**13**	**13**
h2-s	10	**13**	16	**13**	16	16	16	**11**	13
jenkins-s	12	**16**	**16**	**16**	19	19	19	**11**	13
jython-s	9	**13**	**13**	**13**	16	16	16	**9**	13
luindex-s	10	**16**	19	**16**	**16**	19	19	**13**	**13**
lusearch-s	10	**16**	19	**16**	**16**	19	19	**13**	**13**
openjdk8-s	9	**13**	**13**	**13**	16	16	16	**9**	13
pmd-s	11	**16**	19	**16**	**16**	19	19	**13**	**13**
sunflow-s	11	**16**	19	**16**	**16**	19	19	**13**	**13**
tomcat-s	11	**16**	19	**16**	**16**	19	19	**13**	**13**
tradebean-s	11	**16**	19	**16**	**16**	19	19	**13**	**13**
xalan-s	11	**16**	19	**16**	**16**	19	19	**13**	**13**
avrora-l	12	**16**	**16**	**16**	19	19	19	**15**	**15**
batik-l	13	**19**	**19**	**19**	**19**	**19**	**19**	**15**	17
eclipse-l	12	**13**	16	**13**	16	16	16	**11**	15
fop-l	13	**16**	19	**16**	19	19	19	**15**	17
h2-l	13	**16**	**16**	**16**	**16**	**16**	**16**	**15**	**15**
jenkins-l	13	**19**	**19**	**19**	**19**	**19**	**19**	**13**	19
jython-l	13	**16**	19	**16**	19	19	19	**15**	**15**
luindex-l	11	16	**13**	16	16	16	16	**11**	15
lusearch-l	11	16	**13**	16	16	16	16	**11**	15
openjdk8-l	15	**22**	**22**	**22**	**22**	**22**	**22**	**15**	19
pmd-l	12	**16**	**16**	**16**	**16**	**16**	**16**	**15**	**15**
sunflow-l	13	**16**	**16**	**16**	19	**16**	**16**	**15**	**15**
tomcat-l	11	**13**	16	**13**	16	16	16	**11**	15
tradebean-l	12	**13**	16	**13**	16	16	16	**11**	13
xalan-l	12	**16**	**16**	**16**	19	**16**	**16**	**15**	**15**
LiveJournal1-8	6	**13**	**13**	**13**	**13**	**13**	**13**	**9**	**9**
soc-Pokec-8	6	**7**	10	**7**	**7**	10	10	**5**	7
web-BerkStan-8	7	**7**	10	10	10	10	10	**5**	11
web-Google-8	7	**13**	**13**	**13**	16	16	16	**9**	11

compression sequence of the sink-reachability compression operators is always less than 3 times the optimal length.

To answer RQ (4), we note that the best compression sequence to be used in practice is dependent on a trade-off between worst-case asymptotic performance, and observed performance in practice. The sequence $S \circ (D \circ M_i \circ M_c)$ is at most 60% longer than $|C_{opt}|$, except on LiveJournal1-8, where it is 116% longer. This sequence guarantees a maximum length of $3 \times |C_{opt}|$, as all sink-reachability preserving operators are used. However, application of D is not necessary to achieve the same fixpoint, as the equivalence relation of D is a subset of the equivalence relation of $(M_c)^*$ (as per the result in [19]). That is $S \circ (D \circ M_i \circ M_c)^*(G) = S \circ (M_i \circ M_c)^*(G)$. In our experiments, we observe from our experiments that applying D once at the beginning of a sequence leads to a shorter sequence than

using M_i and M_c alone – the last two columns of Table 5.5 show results for $S \circ D \circ (M_i \circ M_c)$ and $S \circ (M_i \circ M_c)$. Using D once is effective, as the sequence $S \circ D \circ (M_i \circ M_c)^*$ is always less than or equal in length to all other reduction sequences in this set of experiments.

The caveat is that the length of $S \circ D \circ (M_i \circ M_c)^*$ is not bounded by $|3 \times C_{opt}|$, and so the worst case performance is substantially more than the best compression sequence with the guarantee of $|3 \times C_{opt}|$ maximum length, being $S \circ (D \circ M_i \circ M_c)$. Hence the best compression sequence to be used in practice is $S \circ D \circ (M_i \circ M_c)^*$, unless the runtime requirements must enforce a strict upper bound, in which case $S \circ (D \circ M_i \circ M_c)$ is best.

5.2.2 Index Construction and Query Processing

The next set of experiments evaluates the effectiveness of the sink-reachability preserving compression framework, in terms of time and space. We perform our evaluation based on the time taken to compute the sink-reachability index,the size of that index, and the time taken to query the sink-reachability of all non-sink vertices in the graph against the index. Our primary motivation is points-to analysis, and so this set of experiments answers points-to aliasing queries, by computing $r_G(u) \cap r_G(v)$, where $u \in V$, $v \in V$, and r is the sink-reachability relation.

As per [19], this can be done by concatenating a graph G with a mirror image of itself G', where G' has one edge $(v', u') \in E'$ for each edge $(u, v) \in E$, and G, G' share the same set of sink-vertices. The operator **X** implements this mirror-transform (see Subsection A3.6.1).

Computing the 2-hop index of [15] on this mirror graph allows aliasing queries to be answered efficiently, in linear time. The operator **I** computes the 2-hop index, while the operator **K** counts the size of this index.

We compare three alternative constructions of the mirror graph in this set of experiments. The first is the mirror graph directly constructed on G_c (the core graph). The second is the mirror graph constructed on G_{fp}, the graph obtained by compressing the graph with sink-reachability preserving compression operators D, M_i, M_c – in the implementation these are operators **D**, **F**, **N**. The third is the mirror graph constructed on G_{ter}, the graph resulting from the bottom-up transitive reduction, followed by the reduction via linear equivalence as introduced in [22]. This third mirror is based on a reduction of the graph which preserves the entire (not just sink) reachability relation, using operators **L** and **B** in the implementation. The results are given in Table 5.6.

Table 5.6 shows index size, construction time, and query time for G_c, G_{fp}, and G_{ter}, with the reported time including the time taken for applying the reduction.

We observe that the reduction itself does not incur significant overhead, because the difference in construction time of the three approaches are negligible on all datasets. The index sizes are greatest for G_c, less for G_{ter}, and least for G_{fp}. This confirms our expectations – as G_c is a precondition for computing both G_{ter} and G_{fp}, and also as G_{fp} computes the sink-reachability index, while G_{ter} computes the (much larger) general reachability index. The compression achieved by sink-reachability preserving operator composition is effective in computing the index, with RQ (5) answered in the affirmative.

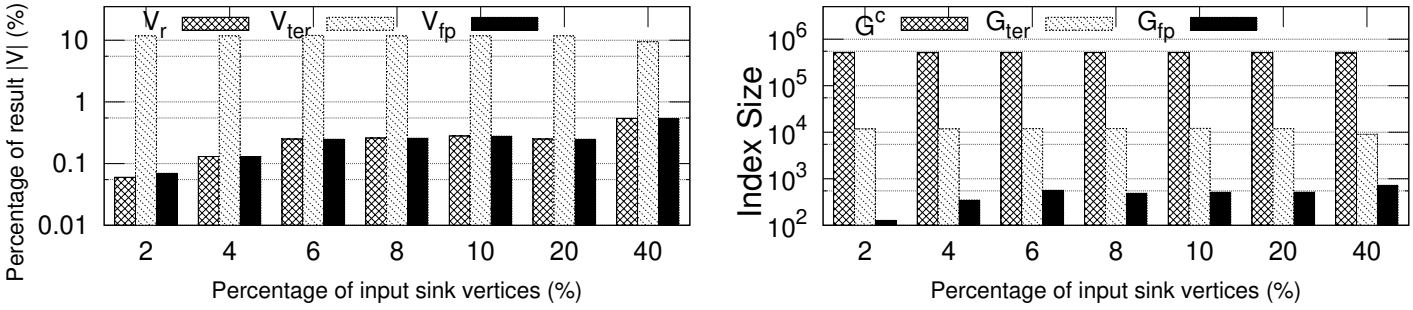
TABLE 5.6. Size, construction time, and query time (in milliseconds) of 2-hop indexes on G^c, on the graph G_{fp} obtained by our fixpoint compression, and on the graph G_{ter} obtained by the compression technique in [87]. The reported construction time includes both graph compression time and index construction time.

	Index on G^c			Index on G_{fp}			Index on G_{ter}		
Graph	Size	Construction (ms)	Query (ms)	Size	Construction (ms)	Query (ms)	Size	Construction (ms)	Query (ms)
avrora-s	654296	3209	19	302515	2950	3	519361	3389	16
batik-s	827578	3973	23	382063	3615	3	656467	4485	19
eclipse-s	379821	1591	12	173130	1547	1	296213	1637	9
fop-s	922000	4539	26	446893	4147	4	735426	5001	21
h2-s	718434	3399	20	306507	2992	3	547584	3544	16
jenkins-s	3287452	24851	107	1493020	22122	18	2450823	26793	81
jython-s	885707	5310	27	386727	4595	4	672469	5440	21
luindex-s	353750	1543	11	160980	1500	1	278685	1525	8
lusearch-s	353031	1620	11	160991	1501	1	278260	1555	9
openjdk8-s	1594631	10642	52	703867	9162	9	1181854	11312	42
pmd-s	667741	3192	19	300372	2960	3	528636	3325	16
sunflow-s	627732	2877	18	285669	2702	3	496825	3030	15
tomcat-s	351841	1506	11	158171	1417	1	276133	1504	9
tradebean-s	313248	1332	10	140248	1245	1	246105	1297	8
xalan-s	736309	3925	24	333058	3626	4	566122	4110	19
avrora-l	2259006	6893	33	379476	5467	3	1274178	6637	34
batik-l	2851914	8458	43	436597	6833	4	1556867	8247	38
eclipse-l	1707429	3209	19	222169	2499	1	1011163	3025	17
fop-l	3366294	10172	42	520894	8111	5	1829424	9928	43
h2-l	2472182	7164	34	383598	5518	3	1366696	6714	31
jenkins-l	20022159	168382	181	1612649	144179	19	6733811	158756	177
jython-l	3477781	65458	44	465887	62039	4	1683750	63601	40
luindex-l	1874032	3199	17	216792	2518	1	1135982	2973	15
lusearch-l	1878680	3142	17	216911	2523	1	1138942	2970	15
openjdk8-l	8499383	32312	87	881799	26299	10	3593158	30702	73
pmd-l	2404183	7071	32	378025	5594	3	1360305	6778	33
sunflow-l	2231046	5962	30	350367	4712	3	1241859	5749	28
tomcat-l	1209228	2922	16	197365	2299	1	695283	2741	15
tradebean-l	1486614	2545	15	179476	1991	1	880781	2402	15
xalan-l	2646944	7818	36	413414	6140	4	1419177	7796	34
LiveJournal1-8	117878	63870	28	85072	42587	3	85222	43398	13
soc-Pokec-8	49970	8251	9	29870	6068	1	30510	6203	1
web-BerkStan-8	1062014	706	6	3091	499	1	27904	516	1
web-Google-8	282703	3986	14	27155	2612	1	73258	2663	8

The advantage of the sink-reachability preserving compression becomes apparent in this result, as the index size is usually the main bottleneck for in-memory reachability query processing [87]. From this, we can expect that compression via the sink-reachability operators will effectively scale query processing and reachability indexing to large graph datasets. Also, note that G_{ter} – the bottom-up transitive reduction followed by the linear equivalence reduction – combined with the reduction provided by the sink-reachability operators to further improve the compression.

5.2.2.1 Varying Percentage of Sink Vertices

Our final set of experiments compares the effectiveness of the sink-reachability preserving operators against the general-reachability preserving operators, in computing the sink-reachability index, when the number of sinks is varied. We perform the evaluation using the large web-BerkStan dataset, with varying percentages of the vertices randomly assigned as sinks using the **A** operator Subsection A3.6.3. The results are shown in the plot below.



(A) Compression effectiveness

(B) 2-hop index size

FIGURE 5.7. Varying number of sink vertices on web-BerkStan

The left plot (a) of Figure 5.7 shows the reduction effectiveness for the reachability kernel, sink-reachability fixpoint, and the techniques in [87]. The y-axis shows the percentage of non-sink vertices retained in the reduction, i.e. $\frac{|V_r|}{|V^c|} \times 100$, $\frac{|V_{ter}|}{|V^c|} \times 100$, $\frac{|V_{fp}|}{|V^c|} \times 100$. The x-axis shows the percentage of sink-vertices randomly assigned as sinks. The right plot (b) of Figure 5.7 shows the size of the 2-hop index for G^c, G_{ter}, and G_{fp} with alternate percentages of sink vertices. The y-axis shows the raw index size as computed by operator **K**, while the x-axis is again the percentage of sink-vertices.

The results in (a) of Figure 5.7 show how the the reduction obtained by G_{ter} is independent of the number of sink-vertices, as G_{ter} computes the general-reachability relation. In contrast, as G_{fp} computes only the sink-reachability relation, its reduction is substantially more effective, most especially when the number of sink vertices is small. Moreover, the reduction achieved by G_{fp} is almost identical to the kernel condensation G_r, which approximates the best possible sink-reachability preserving compression of the graph. The advantage of the sink-reachability preserving operators for sink-reachability queries is clear and strongly evidenced.

On the right, in (b) of Figure 5.7, we observe again that, as expected, the number of sink-vertices does not affect the index size of G^c or G_{ter}. By contrast, the index sizes of the G_{fp} compression are significantly smaller – by several orders of magnitude – than those of G^c and G_{ter}, in all cases. While the index size for G_{fp} does increase with respect to an increase in the percentage of sink-vertices, due to the sink-reachability index being required to capture more information, the growth is negligible, and the reduction scales to higher numbers of sink vertices effectively. This again demonstrates the advantage of the sink-reachability preserving

compression, as index size is the measure most directly correlated with the time and space required to perform reachability queries [19].

This concludes the experimental evaluation of the two systems involved in the hybrid approach.

CHAPTER 6

Related Work

6.1 On Distributed Logic Evaluation

There is an abundance of existing distributed Datalog systems and techniques both in research and industry. These utilize a variety of different distribution models, each similar to, though all ultimately different from, our actor-based streaming method. In this section we review the literature and contextualize our contribution within this related work, and the original techniques presented here are to be taken in contrast to these other approaches.

6.1.1 Logic Programming and Map Reduce

Early distributions of Datalog were instrumented within the then popular map reduce framework, beginning with the work of Afrati *et al*. in 2010 [4] . The distribution is fine-grained, in the sense that each rule evaluation becomes a different map reduce task. This is in contrast to our system, which evaluates collections of rules involved in recursions as separate stratum, assigned to different actors. Each map reduce task is blocking, in that tasks receive inputs strictly at their beginning, perform some computation, and produce outputs strictly at their conclusion – inputs and outputs cannot be streamed.

This was found to be problematic for recursive rules, which are ubiquitous in bottom-up logic programs. A recursive rule evaluated at a current iteration must make use of the results of its evaluation in the previous iteration. To overcome this issue, Afrati *et al*. [4] implement recursive evaluation as a series of map reduce tasks. Each iteration, in contrast to each rule, becomes its own blocking task.

The authors found that runtime was heavily dominated by the communication cost induced by the technique, as well as the overhead of the map reduce system itself. A new task must be created for each recursive round, and any inputs to, or outputs from that task, must be communicated over the network to the next recursive task. In 2011 [5] and 2012 [3], Afrati *et al*. provided optimizations of their technique in two follow up papers, including a means to mitigate the communication cost of intermediate relations between recursive rounds by a distributed hashing technique, and multi-way join taking a single round of communication. While their optimizations demonstrate modest performance improvement, they do not mitigate the primary overheads of communication cost and overheads of the map reduce infrastructure.

Another paper, by Shaw *et al*. in 2012 [66] provides a separate implementation of Datalog in map reduce. This implementation is combined with optimizations of map reduce itself, achieving a 10X performance improvement to the underlying implementation (Apache Hadoop). Even with this optimization, runtime is found to be dominated heavily by the overheads of the map reduce system, with up to 48% of time due to the spent starting map reduce tasks, and up to 500 jobs required to evaluate a single query. Ultimately, map reduce is unsuitable for the sort of recursive computation required for Datalog evaluation.

6.1.2 Logic Programming and Graph Processing Frameworks

An important development in distributed Datalog evaluation was made in 2013, with the work of Seo *et al*. [64]. The authors extend the work of their existing non-distributed Datalog engine, called SociaLite [63], to operate over distributions of machines. Their technique does not rely on any underlying distributed computing framework, and scalable across both nodes and cores thereof.

The distribution method for SociaLite proceeds in a series of epochs, with each epoch computing a stratum, according to the topological ordering of a program's strongly-connected component graph. Independently evaluable, non-recursive rules, are placed into sub-epochs, which are evaluated in parallel. A primary node coordinates the evaluation of these epochs in sequence. An epoch concludes when each worker node signals completion of their workload to the primary node. The distribution is via sharding, either hash or index based, on the first column of a Datalog relation. Data is distributed across worker nodes based on this sharding technique, then across cores of machines further. This is comparable to our combined course-grained plus fine-grained parallelism, however SociaLite offers only limited control over how data is distributed. Our technique is more general, in that the distribution is written as part of the logic program itself, with complex splitting permitting rich customization of actor network topologies.

Authors also include early termination via a bloom filter based approximation, as well as monotone recursive aggregate functions (if an aggregate is monotone, then it admits a fixpoint evaluation, and so can be done within SNE). Early termination via approximation may be contrasted with our system's early production of results. In our design, early production is an inherent property of the evaluation, and not an additional feature, nor entailing any loss whatsoever in precision.

The empirical evaluation of [64] compared two map reduce systems (Hadoop, Haloop) and two graph-based systems (Giraph, Hama). Giraph and Hama use the "think-like a vertex" programming model, which models distributed computation by writing programs as a function executed at each node of a directed graph, with communicating between nodes via edges. This model was introduced in a graph processing framework Pregel, developed at Google in 2010 [45]. Subsequently, Facebook developed their own solution (Giraph), and showed it could be used to process trillion-edge graphs [14] Graph processing has found to be superior to map reduce for many applications, [35] [78], and performance evaluations of think-like a vertex technologies are popular/ubiquitous in the literature. [59] [36] [37] [26] The work in

[64] showed that, on an execution of a single-source shortest path (SSSP) algorithm, Giraph was the clear winner against Hama and the two map reduce frameworks.

SociaLite was then compared directly against Giraph in a series of weak scaling experiments, over six benchmark programs common in social network analysis, using synthetic datasets. Against Giraph on 2 to 64 node instances of synthetic graphs with 8M to 268M vertices, SociaLite performed between 4 and 12 times better than Giraph, and showed near-perfect linear weak scaling on five benchmarks, and within acceptable bounds for the sixth (SSSP).

In 2014, Satish *et al.* [60] compared SociaLite to three non-Datalog based graph processing frameworks – GraphLab, Giraph, and CombBLAS – on four benchmark programs common in graph analysis. These graph processing frameworks are purpose-built for graph analysis algorithms. SociaLite ranked first in one of the benchmarks, and second in all others, with CombBLAS consistently first. This result is impressive for SociaLite, as SociaLite is a general purpose distributed Datalog, while CombBLAS is a highly specialized linear algebra system focused on optimized graph processing algorithms.

In 2016, Moustafa *et al.* [51] implemented a distributed Datalog – called Datalography – within the graph processing framework Giraph. In particular, Datalography uses an asynchronous variant of Giraph, called GiraphUC. GiraphUC has been compared with Giraph under multiple configurations, as well as GraphLab, using four common graph analysis benchmarks, on four real world datasets, with GiraphUC outperforming others by a substantial margin in all cases [26]. Datalography was evaluated against Giraph on three benchmark programs and three real-world datasets. Datalography was up to 9 times faster than native Giraph, and outperformed Giraph on all but one of the nine experiments. Note that each benchmark required 4 lines of Datalog, while the native Giraph code was between 50 to 100 lines of Java.

6.1.3 Logic Programming and Apache Spark

The BigDatalog system, introduced by Shkapsky *et al.* [67], is a distributed Datalog targeting efficient, parallel evaluation, implemented within Apache Spark. Spark is a highly popular, general-purpose data analytics system, designed for large-scale data processing. Spark's approach effectively generalizes map reduce – instead of one map and one reduce operator, however, Spark provides a range of operators (including multiple forms of map, and multiple forms of reduce). Spark, like map reduce, has a blocking execution – in iterative Spark applications, one job is submitted per iteration. In contrast, however, Spark includes many optimizations to the features of map reduce implementations that make it suitable for recursive computations (i.e. such as those required in Datalog rule evaluation). Spark also includes numerous other features, orthogonal to a generalization of map reduce.

BigDatalog programs are compiled to a logical plan, in the same manner as for a typical database query, but with multiple optimization techniques employed (e.g. projection pruning). Non-recursive logical plans are converted to Spark's SQL dialect, and evaluated directly, while recursive logical plans are converted to physical plans over the operators that BigDatalog extends Spark with. The approach is designed for strictly positive programs, that is, Datalog programs that do not involve negation. This avoids the usual ordering imposed by the non-monotonic behaviors of negated literals, and hence the evaluation order is not dependent

on the topological order of program strata. The technique affords greater parallelization, as all relations in a positive program grow monotonically, regardless of the order of rule evaluation. The cost is a significant restriction in language expressiveness, as negation may not be used in BigDatalog programs. This is in contrast to our system, which includes not only negation, but enough expressive power to implement a Turing machine. Favoring efficiency or expressiveness is a choice of design, and a trade-off made based on the target domain of a Datalog engine. While we trade performance for expression, for BigDatalog, the priority is input size and execution speed.

The performance of BigDatalog compared to other systems places it as the current state-of-the-art in distributed Datalog engines. Over 5 common benchmark programs, using both real and synthetic datasets of up to 128 million vertices, BigDatalog outperformed both native Spark programs and SociaLite across all cases. For example, in a 100 million edge evaluation of the transitive closure benchmark, SociaLite ran in 4736 seconds, while BigDatalog took only 22 seconds.

6.1.4 Logic Programming In Industry

Bu *et al*. [9] consider a distributed Datalog from the perspective of the machine learning domain. They consider the distributed computing models of Pregel – a vertex-centric graph processing framework – and iterative map reduce. Their approach involves implementing these distributed computing models within Datalog, then distributing the relational algebra operations of a query plan. Their implementation uses Hyracks, a data-parallel runtime in which a DAG of operators and connectors. In Hyracks, operators perform a computation and connectors redistribute data between operators. Their experimental dataset is of considerable size, with over 1.4 billion vertices in a 70GB snapshot of the worldwide web from Yahoo in 2002. The performance of their system is consistently within acceptable bounds of native Spark and Giraph/Pregel executions.

Chin *et al*. [13] describe a Datalog-like system, developed at Google. The authors give a report of a distributed Datalog built for, and used in, an industrial setting. The system combines both distributed and local computation, whereby rules are transformed into operators over data pipelines. The system supports a batch-based back end for large computations via a transform into map reduce jobs.

While not published in an academic article, in 2012, the Semmle logic engine used to find a mission-critical bug during the 2012 Mars rover mission at NASA JPL [62]. Also outside of academic sphere, the LogicBlox system has been used to generate $150M of savings for a top 10 US retailer [44].

6.2 On Sink-reachability

We now discuss the related work pertaining to our implementation of the sink-reachability preserving framework.

6.2.1 Classical Reachability

Efficiently performing reachability on large-scale graph structured data is a problem of substantial investigation in the literature [79] [84] [12] [85]. The work of Jin *et al.* [30] on their *SCARAB* framework shows that most existing methods to compute the reachability index reach a performance bottleneck, becoming impractical around 1 million vertices and/or edges in either time or resource usage. Yildirim *et al.* [83] have shown that most existing reachability indexation methods are not designed to, nor do they in practice, support large graphs. According to Veloso *et al.* [77], there are only a few systems able to efficiently compute reachability for graphs of greater than 100K edges; in particular, including INTERVAL [54], GRAIL [83], FERRARI [65], TF-Label [12], and FELINE [77].

More recently, the focus has shifted from efficiently computing a reachability index directly, to initially compressing a graph while preserving its reachability relation, then computing the index on this reduced graph. In such systems, a pre-processing stage typically removes all cycles from the graph, using Tarjan's strongly-connected components algorithm [75]. The compression is applied to the acyclic graph, and a pre-processing stage computes the reachability index via the two-hop label introduced by Cohen *et al.* [15]. The two-hop label has been shown to be highly efficient in practice, allowing for reachability queries on the index in near-constant time [79].

6.2.2 Reachability Preserving Compression

Fan *et al.* [22] were the first to propose such a reachability-preserving compression based technique. They introduce the notion of a reduction based on an equivalence relation, where nodes that are equivalent in the original graph by the computed relation are merged together in the transformed graph. The relation considers vertices u and v equivalent in an acyclic graph, if u has the same set of both ancestors and decedents as v. On 10 real-world graphs, with $|V|$ between 2K and 2.4M, and $|E|$ between 21K and 5M, an average compression factor of 95% is obtained. However, the algorithm has high time complexity at $O(|V| \times (|V| + |E|))$, as well as high space complexity, at $O(|V|^2)$.

The work of Zhou *et al.* [87] extends on this approach, by applying an additional transitive reduction compression before compressing via the equivalence relation in [22]. The transitive reduction works in a bottom-up fashion, to remove all redundant transitive edges from a graph G, obtaining a unique minimal transformed graph with the same reachability as G. While the algorithm has worst case time complexity of $O(|V|^3)$, heuristics are employed that are shown to work well in practice, allowing the algorithm to scale performance on large real-world graphs. By applying the transitive reduction initially, the complexity of computing the equivalence relation of [22] is reduced to a time of $O(|V| + |E|)$ and a space of $O(|V|)$ on the transformed graph.

6.2.3 Sink-reachability Preserving Compression

The employment of graph dominators to solve a range of graph problems is ubiquitous in the literature, in particular for flow-graph analysis [75] [40] [10]. Use of dominators for speeding up a points-to analysis first appeared in the work of Nasre *et al.* [53], where it is applied ad-hoc, not as a compression operation. More generally within static analysis, dominator-based techniques have been used in code coverage testing [76]. The work of [19] introduces dominators as a sink-reachability preserving graph compression operation, providing an original algorithm whose implementation is both practical, and, linear in runtime as $O(|V| + |E|)$. It is this algorithm on which we base our implementation of the dominator compression (operator **O**).

Modular decomposition is also a graph algorithm with substantial background in the literature. McConnell *et al.* [48] give the current best time modular decomposition, with a linear time worst case complexity of $O(|V| + |E|)$. However, the algorithm in [48] is of purely theoretical interest, and no known implementation exists in practice. The work of [19] uses a modified variant of modular decomposition as a sink-reachability preserving compression operation. This modified variant works on successor modules only, rather than both successor and predecessor modules as in the original algorithm. This successor-based modular decomposition is instrumented across two operators, M_i and M_c, where obtaining the full modular decomposition requires repeated application of these operators. This is seen as an advantage, as it allows a trade-off between maximum compression and computation efficiency, and, furthermore, permits compositionality with other operators. By interleaving other sink-reachability preserving operators with M_i and M_c, the modular decomposition may be achieved faster and with a more reduced result than by applying M_i and M_c alone repeatedly, until the decomposition is achieved. The [19] includes practical algorithms for M_i and M_c, each running in a linear time of $O(|V| + |E|)$, and algorithms is these on which we base our implementation of the operators **F** and **N**.

6.2.4 Points-to Analysis and Other Applications

Reachability has found extensive application in points-to analysis, in particular for analyses expressed as logic programs [54] [68] [71]. The work of Smaragdakis *et al.* [69] has shown how points-to analysis, implemented as bottom-up logic programs (in Datalog), can be used to compute efficient points-to analyses on the well-known Dacapo benchmarks [7]. Such points-to analyses compute the reachability relation between program variables, and the heap locations they point-to in memory. This problem is the primary motivator of a sink-reachability based analysis, with variables as non-sinks, and heap locations as sinks. By computing only sink-reachability, as opposed to the full reachability, such points-to analyses are a ready target of the compression framework developed here.

Beyond points-to analysis, sink-reachability may be applied to the analysis of links in semantic web documents, social network analysis, access control networks, and large-scale hyperlink graphs. Where previously, these problems have worked on the entire reachability relation, some queries require only the sink-variant, which may be computed substantially more efficiently.

Access control networks model user-permissions over resources as directed, labeled graphs [1]. Treating resources as sinks, users as non-sinks, and rules giving permissions to resources as edges, sink-reachability can be used to efficiently answer queries on resources that users have access to. Reversing the labeling, with users as sinks and resources as non-sinks, allows sink-reachability to express the inverse query of what users may access a resource.

The analysis of influencers in social networks is high value for e-commerce, social media marketing, and brand management [1]. Social networks model users as vertices, and relations between such users (e.g. "friend of") via directed edges. By labeling influencers as sinks, and users as non-sinks, the sink-reachability relation allows one to efficiently answer queries on user-influencer relationships. Queries of this form may be used to identify subsets of users which are not exposed to the content of a particular group of influencers, for the purposes of market saturation in a targeted advertising campaign.

In the Semantic Web, links between web pages or resources are modeled as a complex network. Work on reachability has found substantial application in the analysis of XML documents using ID/IDREF links, and sink-reachability may be applied to speed up a vast range of queries in this domain [85].

In biology, transcriptional regulatory networks are used to model gene relationships as directed graphs. Where one gene regulates the activity of another, the vertices of those two genes have an edge in the graph. Reachability queries have been used to efficiently answer queries of such transition regulatory relationships [46]. As most queries regard selected subsets of genes and relations between them, sink-reachability queries offer substantial opportunities for efficiency, by labeling such subsets as sinks.

CHAPTER 7

Summary

In this work, we have introduced the *hybrid approach*, a solution to the dual problems of distributing logic programs in a practical and general manner, while achieving state of the art performance, on par with native code, for specific, high-impact problems.

In Chapter 3, we have developed a *cloud-based distribution* of logic programs, based on techniques borrowed from stream-programming. This is the first part of the hybrid approach. The distribution offers a new form of parallelism that may be combined with existing parallelization of logic programs. The implementation is a system which offers logic as a service, and is deployable as part of, or readily interfacing with, many existing cloud systems. The distribution is general purpose, and provides a fully automatic synthesis of cloud infrastructure from logic programs. The topology of the network is determined by a program's structure, so by applying program transforms (such as splitting) we may obtain transformed programs which are equivalent in terms of black-box behavior, but dramatically different in terms of structure, and hence distribution also. Our system supports novel extensions, including the use of negation, and the efficient distribution of object data types, that make it useful in practical scenarios.

In Chapter 4, we have given a system which implements a number of *extra-logical algorithms*, which may be incorporated to logic programs (via user-defined functors) to provide performance that is not possible within a standard logic evaluation. This is the second part of the hybrid approach. We have targeted a high-impact problem from the logic programming domain (points-to analysis), and implemented the current state of the art imperative algorithms for computing the bottleneck of this problem (sink reachability). This implementation forms part of the work in [19], a paper I worked on in collaboration with others.

In Chapter 5, we performed an experimental evaluation of both systems involved in the hybrid approach. The first system of the hybrid approach the streaming logic evaluation, is shown to be deployable to the cloud across many machines, with no meaningful performance difference compared to the same distributed evaluation over multiple processes, but performed on a single machine (in Section 5.1).

However, the distributed evaluation, whether across many machines or simulated by multiple processes on one machine, is at a significant performance difference to the non-distributed evaluation. The non-distributed evaluation achieves a multicore, or fine-grained parallelism, which may be combined with the multinode or course-grained parallelism of the distributed evaluation. We have shown the use of a program transformation – the splitting technique – to be effective in increasing parallelism, bringing performance of the distributed evaluation

close to the non-distributed evaluation when both fine-grained and course-grained parallelism are used together for certain benchmarks. Splitting also appears to improve the performance of non-distributed programs, which we conjecture is due to a reduction in the size of the index that must be searched in a rule evaluation.

All experiments compared non-object types (numbers) to object data types (symbols), in order to compare performance of the object distribution, and found no meaningful performance gap between the number and symbol types in any experiment. The system was not generally performant on large-scale datasets. For datasets inducing a large communication volume, the system was shown to perform poorly. For programs above a particular size, the heterogeneity of memory resources at each node required by the distribution, the total resource requirements made computation impractical. Hence, the system is shown to be effective as a cloud-based logic engine, with a multiplicity of novel features which make it useful in practical use cases – but, not as a high-performance distributed evaluation of logic.

The second system involved in the hybrid approach, designed for high-performance on the specific problem of sink-reachability, has been subject to an involved empirical investigation, across numerous real-world, large-scale datasets, drawn from several different problem domains (in Section 5.2). The sink-reachability preserving compression operators are shown to be highly effective, both used alone and in composition. The length of the compression sequences achieved in practice is close to the optimal, and a best-in-practice operator sequence has been evidentially determined. The compression achieved is effective for computing the sink-reachability index, reducing the size of the index, and query time.

The distribution uses a homogeneous allocation of resources (CPU and memory) at each actor. The distribution was unable to evaluate large-sized datasets, due to the resource requirements of certain memory or compute heavy actors being greater than others. As the distribution requires a homogeneous allocation of resources at each node, increasing the memory available to any one memory-intensive actor requires increasing the available memory by the same amount for all actors. As the final R relation of the NR benchmark program used in this experiment requires a join of the n splits $R_1, R_2, \ldots, R_n$, the memory of the actor computing R must be at least $O(\sum_{i=0}^{n} |R_i|)$.

In general, the join actor in the split, which is responsible for merging all split rules, will require memory greater than or equal to the sum of the split relations. Future work, therefore, involves investigation of heterogeneous memory allocation, by providing a static upper bound to the memory usage of a relation at compile time, and taking the maximum of the memory requirements of the relations of an actor as the memory resources to be made available to that actor. Additionally, the same should be done for allocation of compute resources, where the number of threads used in the fine-grained (multi-threaded) parallelism at an actor is determined by a parameter passed to its relations.

The splitting technique speeds up non-distributed evaluation, as evidenced in our experiments. The reason for this is not known, however we have conjectured that it is due to a reduction in the size of the index that must be searched in a rule evaluation. Further research is required in order to determine if this reduction in index size is the cause, and whether by splitting, a bottleneck may be addressed for non-distributed programs. Future direction for this technique

would conceivably involve splits for large programs, for example the DOOP static analysis framework (which is implemented as a logic program).

The system implementing the extra-logical algorithms has not been included in a logic program via user-defined functions, only the method by which to do so presented. The procedure for using the system in real logic programs is straightforward, and so future work includes implementing this inclusion, and using it in large, real-world programs – again, the DOOP static analysis framework presents an feasible and impactful candidate for this.

This concludes our thesis on the hybrid approach for logic programming.

Bibliography

[1] Talel Abdessalem and Imen Ben Dhia. 'A reachability-based access control model for online social networks'. In: *Databases and Social Networks*. 2011, pp. 31–36.

[2] Serge Abiteboul, Richard Hull and Victor Vianu. *Foundations of databases*. Vol. 8. Addison-Wesley Reading, 1995.

[3] Foto N Afrati and Jeffrey D Ullman. 'Transitive closure and recursive datalog implemented on clusters'. In: *Proceedings of the 15th International Conference on Extending Database Technology*. ACM. 2012, pp. 132–143.

[4] Foto N Afrati et al. 'Cluster computing, recursion and datalog'. In: *International Datalog 2.0 Workshop*. Springer. 2010, pp. 120–144.

[5] Foto N Afrati et al. 'Map-reduce extensions and recursive queries'. In: *Proceedings of the 14th international conference on extending database technology*. ACM. 2011, pp. 1–8.

[6] Lars Backstrom et al. 'Group formation in large social networks: membership, growth, and evolution'. In: *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2006, pp. 44–54.

[7] Stephen M Blackburn et al. 'The DaCapo benchmarks: Java benchmarking development and analysis'. In: *ACM Sigplan Notices*. Vol. 41. 10. ACM. 2006, pp. 169–190.

[8] Paolo Boldi and Sebastiano Vigna. 'The webgraph framework I: compression techniques'. In: *Proceedings of the 13th international conference on World Wide Web*. 2004, pp. 595–602.

[9] Yingyi Bu et al. 'Scaling datalog for machine learning on big data'. In: *arXiv preprint arXiv:1203.0160* (2012).

[10] Adam L Buchsbaum et al. 'Linear-time algorithms for dominators and other path-evaluation problems'. In: *SIAM Journal on Computing* 38.4 (2008), pp. 1533–1573.

[11] Stefano Ceri, Georg Gottlob and Letizia Tanca. '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.

[12] James Cheng et al. 'TF-Label: a topological-folding labeling scheme for reachability querying in a large graph'. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 2013, pp. 193–204.

[13] Brian Chin et al. 'Yedalog: Exploring knowledge at scale'. In: *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2015.

[14] Avery Ching et al. 'One trillion edges: Graph processing at facebook-scale'. In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1804–1815.

[15] Edith Cohen et al. ‘Reachability and distance queries via 2-hop labels’. In: *SIAM Journal on Computing* 32.5 (2003), pp. 1338–1355.

[16] Don Coppersmith and Shmuel Winograd. ‘Matrix multiplication via arithmetic progressions’. In: *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. ACM. 1987, pp. 1–6.

[17] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009.

[18] Jens Dietrich, Nicholas Hollingum and Bernhard Scholz. ‘Giga-scale exhaustive points-to analysis for java in under a minute’. In: *ACM SIGPLAN Notices*. Vol. 50. 10. ACM. 2015, pp. 535–551.

[19] Jens Dietrich et al. ‘Efficient Sink-Reachability Analysis via Graph Reduction’. In: *IEEE Transactions on Knowledge & Data Engineering* 01 (2021), pp. 1–1.

[20] Jens Dietrich et al. ‘Evil pickles: DoS attacks based on object-graph engineering’. In: *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017.

[21] Pranay Dutta and Prashant Dutta. ‘Comparative Study of Cloud Services Offered by Amazon, Microsoft & Google’. In: *International Journal of Trend in Scientific Research and Development* 3.3 (2019), pp. 981–985.

[22] Wenfei Fan et al. ‘Query preserving graph compression’. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 2012, pp. 157–168.

[23] Tibor Gallai. ‘Transitiv orientierbare graphen’. In: *Acta Mathematica Hungarica* 18.1-2 (1967), pp. 25–66.

[24] Sergio Greco and Cristian Molinaro. ‘Datalog and logic databases’. In: *Synthesis Lectures on Data Management* 7.2 (2015), pp. 1–169.

[25] Todd J Green et al. ‘Datalog and recursive query processing’. In: *Foundations and Trends® in Databases* 5.2 (2013), pp. 105–195.

[26] Minyang Han and Khuzaima Daudjee. ‘Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems’. In: *Proceedings of the VLDB Endowment* 8.9 (2015), pp. 950–961.

[27] Carl Hewitt. ‘Actor model of computation: scalable robust information systems’. In: *arXiv preprint arXiv:1008.1459* (2010).

[28] Mark D Hill and Michael R Marty. ‘Amdahl’s law in the multicore era’. In: *Computer* 41.7 (2008), pp. 33–38.

[29] Xiaowen Hu et al. ‘An Efficient Interpreter for Datalog by De-Specializing Relations’. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation* (2021), tbd.

[30] Ruoming Jin et al. ‘SCARAB: scaling reachability computation on large graphs’. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM. 2012, pp. 169–180.

[31] Herbert Jordan, Bernhard Scholz and Pavle Subotić. ‘Soufflé: On synthesis of program analyzers’. In: *International Conference on Computer Aided Verification*. Springer. 2016, pp. 422–430.

[32] Herbert Jordan et al. ‘A specialized B-tree for concurrent datalog evaluation.’ In: *PPoPP*. 2019, pp. 327–339.

[33] Herbert Jordan et al. 'Brie: A Specialized Trie for Concurrent Datalog'. In: *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*. ACM. 2019, pp. 31–40.

[34] Herbert Jordan et al. 'Specializing parallel data structures for Datalog'. In: *Concurrency and Computation: Practice and Experience* (2020), e5643.

[35] Tomasz Kajdanowicz et al. 'Comparison of the efficiency of mapreduce and bulk synchronous parallel approaches to large network processing'. In: *2012 IEEE 12th International Conference on Data Mining Workshops*. IEEE. 2012, pp. 218–225.

[36] Vasiliki Kalavri, Vladimir Vlassov and Seif Haridi. 'High-level programming abstractions for distributed graph processing'. In: *IEEE Transactions on Knowledge and Data Engineering* 30.2 (2017), pp. 305–324.

[37] Jannis Koch et al. 'An empirical comparison of big graph frameworks in the context of network analysis'. In: *Social Network Analysis and Mining* 6.1 (2016), p. 84.

[38] Ajay D Kshemkalyani and Mukesh Singhal. *Distributed computing: principles, algorithms, and systems*. Cambridge University Press, 2011.

[39] André Langevin and Diane Riopel. *Logistics Systems: Design and Optimization*. Springer, Jan. 2005, p. 388. ISBN: 978-0-387-24977-3.

[40] Thomas Lengauer and Robert Endre Tarjan. 'A fast algorithm for finding dominators in a flowgraph'. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1.1 (1979), pp. 121–141.

[41] Jure Leskovec, Daniel Huttenlocher and Jon Kleinberg. 'Predicting positive and negative links in online social networks'. In: *Proceedings of the 19th international conference on World wide web*. 2010, pp. 641–650.

[42] Jure Leskovec and Andrej Krevl. 'SNAP datasets: Stanford large network dataset collection (2014)'. In: *URL http://snap. stanford. edu/data* (2016), p. 49.

[43] Jure Leskovec et al. 'Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters'. In: *Internet Mathematics* 6.1 (2009), pp. 29–123.

[44] LogicBlox. *Customer Demand Modelling*. https://developer.logicblox.com/consumer-demand-modeling. Accessed: 2019-10-05.

[45] Grzegorz Malewicz et al. 'Pregel: a system for large-scale graph processing'. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, pp. 135–146.

[46] Oliver Mason and Mark Verwoerd. 'Graph theory and networks in biology'. In: *IET systems biology* 1.2 (2007), pp. 89–119.

[47] Julian J McAuley and Jure Leskovec. 'Learning to discover social circles in ego networks.' In: *NIPS*. Vol. 2012. Citeseer. 2012, pp. 548–56.

[48] Ross M McConnell and Fabien De Montgolfier. 'Linear-time modular decomposition of directed graphs'. In: *Discrete Applied Mathematics* 145.2 (2005), pp. 198–209.

[49] Frank McSherry et al. 'Composable incremental and iterative data-parallel computation with naiad'. In: *Microsoft Research* (2012).

[50] Frank McSherry et al. 'Differential Dataflow.' In: *CIDR*. 2013.

[51] Walaa Eldin Moustafa et al. 'Datalography: Scaling datalog graph analytics on graph processing systems'. In: *2016 IEEE International Conference on Big Data (Big Data)*. IEEE. 2016, pp. 56–65.

[52] Patrick Nappa et al. 'Fast Parallel Equivalence Relations in a Datalog Compiler'. In: *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 2019, pp. 82–96.

[53] Rupesh Nasre. 'Exploiting the structure of the constraint graph for efficient points-to analysis'. In: *ACM SIGPLAN Notices*. Vol. 47. 11. ACM. 2012, pp. 121–132.

[54] E Nuutila. *Efficient Transitive Closure Computation in Large Digraphs, Mathematics and Computing in Engineering Series No. 74 PhD thesis Helsinki University ofTechnology*. 1995.

[55] Rabi Prasad Padhy, Manas Ranjan Patra and Suresh Chandra Satapathy. 'X-as-a-Service: Cloud Computing with Google App Engine, Amazon Web Services, Microsoft Azure and Force. com'. In: *Com. Int. J. Comput. Sci. Telecommun* 2.9 (2011).

[56] Sriram Raghavan and Hector Garcia-Molina. 'Representing web graphs'. In: *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*. IEEE. 2003, pp. 405–416.

[57] Shawn Rasheed, Jens Dietrich and Amjed Tahir. 'Laughter in the Wild: A Study Into DoS Vulnerabilities in YAML Libraries'. In: *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE. 2019, pp. 342–349.

[58] Thomas Reps. 'Program analysis via graph reachability'. In: *Information and software technology* 40.11-12 (1998), pp. 701–726.

[59] Sherif Sakr. 'Processing large-scale graph data: A guide to current technology'. In: *IBM Developerworks* (2013), p. 15.

[60] Nadathur Satish et al. 'Navigating the maze of graph analytics frameworks using massive graph datasets'. In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM. 2014, pp. 979–990.

[61] Bernhard Scholz et al. 'On fast large-scale program analysis in datalog'. In: *Proceedings of the 25th International Conference on Compiler Construction*. ACM. 2016, pp. 196–206.

[62] Semmle. *Semmle at NASA: Landing Curiosity safely on Mars*. https://semmle.com/case-studies/semmle-nasa-landing-curiosity-safely-mars. Accessed: 2019-10-05.

[63] Jiwon Seo, Stephen Guo and Monica S Lam. 'SociaLite: Datalog extensions for efficient social network analysis'. In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE. 2013, pp. 278–289.

[64] Jiwon Seo et al. 'Distributed socialite: a datalog-based language for large-scale graph analysis'. In: *Proceedings of the VLDB Endowment* 6.14 (2013), pp. 1906–1917.

[65] Stephan Seufert et al. 'Ferrari: Flexible and efficient reachability range assignment for graph indexing'. In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE. 2013, pp. 1009–1020.

[66] Marianne Shaw et al. 'Optimizing large-scale Semi-Naïve datalog evaluation in hadoop'. In: *International Datalog 2.0 Workshop*. Springer. 2012, pp. 165–176.

[67] Alexander Shkapsky et al. 'Big data analytics with datalog queries on spark'. In: *Proceedings of the 2016 International Conference on Management of Data*. ACM. 2016, pp. 1135–1149.

[68] Yannis Smaragdakis, George Balatsouras et al. 'Pointer analysis'. In: *Foundations and Trends® in Programming Languages* 2.1 (2015), pp. 1–69.
[69] Yannis Smaragdakis and Martin Bravenboer. 'Using Datalog for fast and easy program analysis'. In: *International Datalog 2.0 Workshop*. Springer. 2010, pp. 245–251.
[70] Yannis Smaragdakis and Martin Bravenboer. 'Using Datalog for fast and easy program analysis'. In: *Datalog Reloaded*. Springer, 2011, pp. 245–251.
[71] Manu Sridharan et al. 'Demand-driven points-to analysis for Java'. In: *ACM SIGPLAN Notices*. Vol. 40. 10. ACM. 2005, pp. 59–76.
[72] Volker Strassen. 'Gaussian elimination is not optimal'. In: *Numerische mathematik* 13.4 (1969), pp. 354–356.
[73] Pavle Subotić et al. 'Automatic index selection for large-scale datalog computation'. In: *Proceedings of the VLDB Endowment* 12.2 (2018), pp. 141–153.
[74] Lubos Takac and Michal Zabovsky. 'Data analysis in public social networks'. In: *International scientific conference and international workshop present day trends of innovations*. Vol. 1. 6. 2012.
[75] Robert Tarjan. 'Depth-first search and linear graph algorithms'. In: *SIAM journal on computing* 1.2 (1972), pp. 146–160.
[76] Mustafa M Tikir and Jeffrey K Hollingsworth. 'Efficient instrumentation for code coverage testing'. In: *ACM SIGSOFT Software Engineering Notes*. Vol. 27. 4. ACM. 2002, pp. 86–96.
[77] Renê Rodrigues Veloso et al. 'Reachability Queries in Very Large Graphs: A Fast Refined Online Search Approach.' In: *EDBT*. Citeseer. 2014, pp. 511–522.
[78] CL Vidal-Silva et al. 'Advantages of Giraph over Hadoop in Graph Processing'. In: *Engineering, Technology & Applied Science Research* 9.3 (2019), pp. 4112–4115.
[79] Haixun Wang et al. 'Dual labeling: Answering graph reachability queries in constant time'. In: *22nd International Conference on Data Engineering (ICDE'06)*. IEEE. 2006, pp. 75–75.
[80] Lizhe Wang et al. 'Cloud computing: a perspective study'. In: *New generation computing* 28.2 (2010), pp. 137–146.
[81] Jaewon Yang and Jure Leskovec. 'Defining and evaluating network communities based on ground-truth'. In: *Knowledge and Information Systems* 42.1 (2015), pp. 181–213.
[82] Jaewon Yang and Jure Leskovec. 'Patterns of temporal variation in online media'. In: *Proceedings of the fourth ACM international conference on Web search and data mining*. 2011, pp. 177–186.
[83] Hilmi Yildirim, Vineet Chaoji and Mohammed J Zaki. 'Grail: Scalable reachability index for large graphs'. In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 276–284.
[84] Hilmi Yıldırım, Vineet Chaoji and Mohammed J Zaki. 'GRAIL: a scalable index for reachability queries in very large graphs'. In: *The VLDB Journal* 21.4 (2012), pp. 509–534.
[85] Jeffrey Xu Yu and Jiefeng Cheng. 'Graph reachability queries: A survey'. In: *Managing and Mining Graph Data*. Springer, 2010, pp. 181–215.
[86] Junfeng Zhou et al. 'Accelerating reachability query processing based on DAG reduction'. In: *The VLDB Journal—The International Journal on Very Large Data Bases* 27.2 (2018), pp. 271–296.

[87] Junfeng Zhou et al. ‘DAG reduction: Fast answering reachability queries’. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM. 2017, pp. 375–390.

APPENDIX A

Documentation of Sink-Reachability Framework

A1 Command Line Interface

The available functionality of the sink-reachability preserving graph compression framework is captured in its command line interface (or CLI). The options available in the CLI are presented in Table A.1 below.

TABLE A.1. Command line interface of graph processing system.

-a	-algorithms	<OP>(<OP>)*
-b	-block	<OP>(<OP>)*
-c	-count	[0-9]+ | min | max
-d	-debug	
-e	-extra	.*=.*(,.*=.*)*
-f	-files	
-i	-inputs	FILE(,<FILE>)*
-j	-jobs	all | [0-9]+
-l	-log	
-o	-output	<DIR>
-p	-prefix	<OP>(<OP>)*
-s	-suffix	<OP>(<OP>)*
-v	-verify	
-h	-help	

The leftmost column is the short-form option, the center column is the long-form option, and the rightmost column is a BNF-style description of the format of the arguments an option accepts. These options control the behavior of the system for a given execution of the system, in terms of what graphs are processed, and how they are processed. An execution involves reading a graph or several graphs from input files, applying one or many graph operations upon those graphs, and writing results of the transformations applied to output files.

A1.1 Input Files

The `-i` or `-inputs` option takes a list of comma-separated file paths as an argument. The file at each path must be in a graph format, with each line having either one vertex label,

denoting a vertex, or two vertex labels separated by a space, denoting two vertices and a directed edge from the first to the second. Vertex label beginning with an `h` are treated as sinks, otherwise they are treated as non-sink or *normal* vertices. The `-j` or `-jobs` option allows each input graph to be processed on a different thread, with either a numeric argument giving the number of threads, or `all` to use one thread per file.

A1.2 Graph Operators

There are several options involved in defining the sequences of operators that are run by the system on the graph during an execution. In the arguments to these options, operators are each identified by a single letter, for example `S` is the operator for SCC, while `D` is the operator for dominator. An *operator sequence* is a string of operator identifiers that specifies a series of operations to be performed on a graph, so the operator sequence `SD` would apply the SCC compression followed by the dominator compression on the input graph or graphs.

The `-a` or `-algorithms` option takes an operator sequence, and then applies each operator in this sequence to the input graph according to the arguments to the `-c` or `-count` option. If `-count` is passed an integer argument, then this argument specifies the number of operations performed on the graph. Where `-algorithms` is used with `-count`, the next operator is selected in a round-robin fashion on the operator sequence supplied. So if we use `-a` $A_1A_2 \dots A_m$ `-c` n, then for i from 1 to n, we apply $A_{i \bmod m}$. If `-count` is `min`, then operators are repeatedly applied to the graph until no further operator application reduces the graph, and so the execution has reached a fixpoint. If `-count` is `max`, the system finds the first operator sequence reaching a fixpoint, then repeatedly searches for operator sequences less than this length that also reach a fixpoint. Where a lesser-length fixpoint operator sequence is found, the search continues for sequences with a lesser size. The search concludes when all operator sequences with a size less than the current minimum-length fixpoint operator sequence are found to *not* reach the fixpoint.

The `-b` or `-block` option is similar to `-algorithms`, however the count is not applied to each operator in the sequence in a round-robin fashion, but to the sequence as a whole. So if we use `-b` $B_1B_2 \dots B_m$ `-c` n, we set $i = 0$, apply $B_1, B_2, \dots, B_m$, update i as $i = i+m$, and repeat until $i \geq n$, or a fixpoint is reached. The `-p` or `-prefix` option specifies a sequence of operators to be run before fixpoint computation. This is useful for allowing algorithms which have no further effect on the graph to provide pre-processing, for example instead of passing `SD` to `-algorithms`, we can pass `S` to `-prefix` and `D` to `-algorithms` as `S` has no effect on a graph resulting from applying `D`. The `-s` or `-suffix` option is similar to `-prefix`, but allows operator sequences to be applied after a fixpoint has been reached, or a count (via `-count`) has been exceeded. This is useful for performing non-reducing operations on the graph. For example, the operator `K` may be used to construct and compute the size of the two-hop label, in order to compare the size of the index resulting from different reductions.

The `-e` or `-extra` option allows additional configuration to be passed to the operators, and takes a set of key value pairs in the format `k1=v1,k2=v2,...`. For example, the `Y` option randomly assigns sink vertices to a graph, and using

`-extra sinksOnly=1,sample=25.0,population=50.0` causes a random sample of 25.0%, of a population made up of 50.0% of the sinks in the input graph, to be used as the sinks in the output graph.

A brief description of the set of operator identifiers is given below, a later section will give detailed information on the data structures and algorithms employed in their computation.

O: Read the graph from a file, removing duplicate edges.
C: Add a root vertex with an edge from each sink vertex to that vertex; this allows for later processing via other algorithms.
S: Apply the SCC compression on the graph, using Tarjan's algorithm in [75], with the root created by applying `C`, and edges backwards directed.
T: Remove all vertices that are not reachable in a DFS from the root vertex on the backwards-directed graph.
D: Apply the dominator compression, as described in [19].
F: Apply the M_i modular decomposition compression, as described in [19].
N: Apply the M_c modular decomposition compression, as described in [19].
B: Apply the bottom-up transitive reduction, as described in [86].
L: Apply the linear equivalence reduction, as described in [22].
V: Compute the vertex kernel, as in [19].
I: Compute the two-hop label, as described in [15].
K: Get the size of the two-hop label, as computed by `I`.
X: Construct the mirror graph in [19]; this is used to answer aliasing queries in points-to analysis.
A: Randomly assign a selection of vertices in the graph as sinks, with the selection determined by arguments passed to `-extra`.

A1.3 Output Files

The `-o` or `-output` option takes a directory as an argument, to which the results of an execution will be written as output files.

The `-l` or `-log` option is a flag, taking no arguments, which enables logging of graph statistics for an evaluation. A set of statistics for an evaluation is logged for each application of a graph operator to an input graph. The log file contains statistics for all operations performed on all input graphs of an execution, with one set of statistics per line. The log file is in CSV format, with the following list describing the fields (columns) of each line.

GraphName: The name of the input graph.
AlgorithmRegex: A regular expression, describing the operator sequence resulting from the `-algorithms`, `-block`, `-prefix`, `-suffix`, and `-count` options.
AlgorithmSequenceLength: The length of the algorithm sequence applied to the graph corresponding to this set of statistics.
TotalVertices: The total number of root, sink, and normal (i.e. non-sink, non-root) vertices.
NormalVertices: The number of vertices which are not root or sink vertices.
SinkVertices: The number of sink vertices.

RootVertices: The number of root vertices, as added by application of the `C` operator; this may be either 0 or 1.
TotalEdges: The total number of edges in the graph.
NormalEdges: The number of edges which have only normal (non-sink, non-root) vertices as their endpoints.
SinkEdges: The number of edges which have a sink as their endpoint.
RootEdges: The number of edges which have the root as their endpoint.
IsReversed: True if the edges are backwards directed, i.e. from the sinks to the normals, false otherwise.
IsAcyclic: True if the graph contains no cycles, false otherwise.
IsTerminated: True if a fixpoint has been reached for the graph corresponding to this set of statistics, false otherwise.
CurrentRuntimeMs: The current time in milliseconds since the start of execution at the point where the log message was emitted.
TwoHopLabelSize: The size of the two-hop label, obtained by applying operator `K`, and 0 if this operator has not been applied.

The `-f` or `-files` option allows files to be written in `.facts` file format for use with Souffle Datalog programs, with one set of fact files written per operator application, per input graph.

A1.4 Verification and Debugging

The `-v` or `-verify` option verifies the sink-reachability relation of the transformed graph against the sink-reachability relation of the input graph, by comparing the two-hop labels of both as computed by operator `I`.

The `-d` or `-debug` option writes a set of debug files for each application of each operator to a graph, including CSV files of all vertices, sinks, roots, edges, equivalence relations computed between vertices by operators, and an description of the graph in Graphviz DOT format which may be viewed as an image.

Finally, the `-h` or `-help` option provides information on each of the CLI options, in a format similar to Table A.1 presented at the beginning of this section.

A2 Implementation of Data Structures and Algorithms

This section covers the data structures and algorithms used in the sink-reachability framework.

The fundamental data structure of the system is a graph, while the algorithms are the operators, and functions available on the graph used in the operators. Each operator application transforms the graph, typically merging vertices and edge sets via an equivalence-based reduction. The graph preserves a mapping from vertices in the original graph to their equivalence classes, such that reachability relationships between vertices in the original graph may be obtained by resolving them to the representative vertices of their equivalence classes.

We contribute pseudocode for all but the most trivial algorithms used in our system, with detailed descriptions of the procedure each expresses. For the sake of brevity, the pseudocode and descriptions are placed in the appendix, and referred to here only as matter additional to, not mandatory for, a reading of this section.

In implementation, the graph is a tuple $G = (L, D, A)$. L is a list of vertex-labels, where the index of a vertex-label in this list acts as a unique identifier for that vertex. D is a disjoint-set data structure mapping the unique identifier of every vertex to the identifier of the representative of the equivalence class in which it is contained. A is an adjacency list data structure, with one sub-list per representative of an equivalence class in the disjoint-sets data structure, and each sub-list containing a sequence of unique vertex identifiers representing edge endpoints.

The available behaviors of G and its fields are described hence.

A2.1 Vertex Labels $G.L$

The vertex label list $G.L$ is implemented with a C++ `std::vector` of `std::string` types. $G.L$ has standard operations for access, with $G.L[i]$, and append, with $G.push\_back(v)$. The index of a vertex label in $G.L$ is the unique identifier of that vertex. Vertex labels are classified as either root, sink, or normal vertices, depending on whether they begin with an `r`, `h`, or other character respectively. Additional operations are defined on checking the class of a vertex for convenience, with $u \in G.R$, $u \in G.S$, and $u \in G.V$ used to denote if a vertex u is a root, sink, or normal vertex respectively. Where u is a vertex label, u is checked directly by comparison on its first character, while if u is a unique identifier for a vertex, it is resolved as an index and the label $G.L[u]$ is checked.

A2.2 Adjacency List $G.A$

The adjacency list represents the edge set of the graph, with one sub-list for each unique id of a disjoint set representative. Hence, if $G[i][j] = k$, then there is an edge from the vertex with identifier i to the vertex with identifier k in the graph. The adjacency list $G.A$ supports a number of standard list (C++ `std::vector`) functions, such as $G.A[i]$ for access of the ith sub-list of $G.A$, $G.A[i][j]$ for the jth member of the ith sub-list, $G.A[i]$.push_back(k) to append k to the sub-list $G[i]$, among others. Beyond the standard vector operations, a number

of graph-specific functions are defined to operate over the adjacency lists of graphs, described as follows.

A2.2.1 TOPOLOGICALORDER

The TOPOLOGICALORDER algorithm computes a topological order over the vertices of the graph.

Input: $G.A, u$
Output: o
$o \leftarrow$ list of size $|G.A|$ s.t. $\forall 0 \leq i < |G.A| : o[i] = \perp$
$c \leftarrow 0$
Function ACYCLICDFS(i)**:**
  if $o[i] = \perp$ **then**
    for $j \in G.A[i]$ **do**
      $f(j)$
    end
    $o[i] \leftarrow c$
    $c \leftarrow c + 1$
  end
ACYCLICDFS(u)
$k \leftarrow c - 1$
for $0 \leq i < |o|$ **do**
  if $o[i] \neq \perp$ **then**
    $o[i] \leftarrow k - o[i]$
  end
end
return o

Algorithm 9: TOPOLOGICALORDER

The algorithm takes as input the adjacency list of the graph, $G.A$, and a designated root vertex u. Note that this assumes the graph in $G.A$ is acyclic, otherwise the top-sort is not defined. The output is a mapping o, where if $o[v] = i$ then the vertex v has the index i in the top-sort of $G.A$.

The algorithm has a simple recursive DFS implementation, which visits each vertex of $G.A$ in a topological order, updating o during the search. The remaining unvisited vertex are then added to the topological order in an arbitrary order. The algorithm assumes that such remaining unvisited vertices are irrelevant to the order, as they are not reachable from the root.

The algorithm has worst-case $O(|V| + |E|)$ time complexity, as per the linear runtime of the DFS search.

A2.2.2 TRANSPOSE

The TRANSPOSE algorithm reverses all edges in the graph, by transposing the adjacency list $G.A$.

Input: $G.A$
Output: B
$B \leftarrow$ adjacency list of size $|G.A|$
for $0 \leq i < |G.A|$ **do**
  for $0 \leq j < |G.A[i]|$ **do**
    $k \leftarrow G.A[i][j]$
    $B[k]$.push_back(i)
  end
end
return B

Algorithm 10: TRANSPOSE

The algorithm returns a new adjacency list, with one new edge (v, u) at $G[v][u]$ for each existing edge (u, v) at $G[u][v]$. The returned adjacency list may then be subsequently assigned to $G.A$ to update the graph. The algorithm has $O(|E|)$ time complexity, as it performs a constant number of operations per each edge of G.

A2.2.3 DFS

The DFS algorithm is a simple iterative, stack-based DFS implementation, returning each vertex visited from a given root in the order encountered. It handles cycles, and a result is defined regardless of the acyclicity of the graph. The runtime is linear, with a worst case $O(|V| + |E|)$.

A2.2.4 DFSORDER

The DFSORDER function is the same as the DFS algorithm above, however it returns a map from a vertex to its index in the order. The runtime is linear, with a worst case $O(|V| + |E|)$.

A2.3 Disjoint Sets $G.D$

The disjoint sets data structure of the graph $G.D$ maps each input vertex to a representative vertex of the equivalence class to which it belongs. All vertices (including sinks) are initially in their own equivalence classes. Application of graph operators induce unifications of these equivalence classes, whereby which equivalence classes are merged together. All vertices in an equivalence class resolve to the same representative, specifically the vertex with the minimum unique identifier in the equivalence class.

The implementation relies on an array-backed list $G.ds$, which is equivalent in size to $G.L$, and if $G.ds[u] = v$ then u belongs to the equivalence class of v. The operations on the disjoint sets data structure are ubiquitous in the remainder of this section, and so are typically abbreviated

to notation for convenience. For example, $G.D(u)$ is implemented as the GETMEMBERS explained below. A description of available functions on the disjoint sets $G.D$ is given hence.

A2.3.1 FINDREP

The FINDREP algorithm returns the representative of the equivalence class to which the input vertex belongs. It is notated herein, with $G.D[u] = v$ denoting that v is the representative of the equivalence class to which both u and v belong.

Input: G, u
Output: $G.D[u]$
if $G.ds[u] = u$ **then**
  return u
else
  $G.ds[u] = G.D[G.ds[u]]$
  return $G.ds[u]$
end

Algorithm 11: FINDREP

The algorithm first checks if the given vertex u is the representative of itself, and hence it's equivalence class in $G.ds$. If so, u is returned. If not, then the representative of u in $G.ds$ is assigned to the representative of the recursive call of the FINDREP algorithm on $G.ds[u]$, in the statement $G.ds[u] = G.D[G.ds[u]]$; then $G.ds[u]$ returned. The algorithm has (average case) constant time complexity.

A2.3.2 UNIFYSETS

The UNIFYSETS function unifies the equivalence classes of two given unique vertex identifiers u, v in G.

Input: G, u, v
if $u \neq v$ ***or*** $G.D[u] = G.D[v]$ **then**
  if $G.D[u] < G.D[v]$ **then**
    $G.ds[v] = u$
  else
    $G.ds[u] = v$
  end
end

Algorithm 12: UNIFYSETS

The algorithm first checks that the identifiers are not already in the same equivalence class. The algorithm compares the representatives of the identifiers, and uses the minimum representative to determine the equivalence class to which both are to be merged. Thus, $G.ds[u], G.ds[v]$ are assigned to the minimum of $G.D[u]$ and $G.D[v]$.

A2.3.3 COUNTREPS

The COUNTREPS operation gives the number of disjoint set representatives, that is, the number of members of $G.ds[u]$ s.t. $G.ds[u] = u$, which is equal also to the number of equivalence classes. This function is notated in later algorithms as $|G.D|$.

Input: G
Output: $|G.D|$
1 $c \leftarrow 0$
2 **for** $0 \leq i < |G.ds|$ **do**
3 **if** $G.ds[i] = G.D[i]$ **then**
4 $c \leftarrow c + 1$
5 **end**
6 **end**
7 **return** c

Algorithm 13: COUNTREPS

The algorithm performs a linear scan over each unique identifier, checking whether it is the representative of its equivalence class, and incrementing the counter if so. The runtime of this algorithm is hence linear, being $O(|G.ds|)$, which is the same as $O(|G.L|)$.

A2.3.4 GETMEMBERS

The GETMEMBERS function returns all unique identifiers that have the same equivalence class as the given vertex. The notation $G.D(u)$ is used herein to denote a call to this function.

Input: G, u
Output: $G.D(u)$
1 $U \leftarrow$ empty list
2 **for** $0 \leq i < |G.ds|$ **do**
3 **if** $G.D[G.ds[i]] = G.D[u]$ **then**
4 U.push_back(i)
5 **end**
6 **end**
7 **return** U

Algorithm 14: GETMEMBERS

The algorithm iterates over each unique vertex identifier in $G.ds$, and returns the list composed of all such unique identifiers u such that $G.D[G.ds[u]] = G.D[u]$. This algorithm is the same in asymptotic runtime to a linear scan on $G.ds$, which is $O(|G.ds|)$ or (equivalently) $O(|G.L|)$.

A2.3.5 GETALLREPS

The GETALLREPS algorithm returns a list U, containing all disjoint set representatives. This is called during iterations over $G.D$ in subsequent algorithms, with notation such as $\forall u \in G.D$ or **for** $u \in G.D$ **do** used to iterate over all disjoint set representatives.

```
   Input: G
   Output: G.D
1  U ← empty list
2  for 0 ≤ i < |G.ds| do
3  |  if i = G.D[i] then
4  |  |  U.push_back(i)
5  |  end
6  end
7  return U
```

Algorithm 15: GETALLREPS

The algorithm iterates over each unique identifier, checking if that identifier is a representative in $i = G.D[i]$, and, if so, adding it to the output list U. As this algorithm is also the same in asymptotic runtime to a linear scan on $G.ds$, it has a time of $O(|G.ds|)$ or (equivalently) $O(|G.L|)$.

A2.4 Graph Mutation Algorithms

The next set of data-structure operations are functions which mutate the graph as a whole. Only two of the following algorithms are used in the rest of the code SORTGRAPHBYLABELS, and SORTGRAPHBYREPS, while the others are used within these algorithms. Both SORTGRAPHBYLABELS and SORTGRAPHBYREPS apply a linear-time transform on the graph, establishing a series of assumptions over the graph's state that are utilized by other algorithms, where these assumptions afford substantial gains in both time and space efficiency.

A2.4.1 UNIFYGRAPH

The UNIFYGRAPH algorithm applies the equivalence relation of the vertices in the disjoint sets $G.D$ to the edges of the adjacency list $G.A$. When the equivalence classes of u and v are merged in a call to UNIFYSETS, all vertices of u and v are conceptually merged into the same vertex of the graph. The edge sets of the vertices in the equivalence classes must also then be merged, otherwise $G.D$ and $G.A$ will be inconsistent – this is the responsibility of the UNIFYGRAPH operation.

We delay the merge on the edge sets in $G.A$ until an explicit call to this function, so that a unification of u and v in $G.D$ does not automatically induce a merge of the corresponding edge sets of $G.A$. The unification of disjoint sets is constant time in the average case, while the merge of the edge sets is, in the worst case linear as $O(|E|)$. By having the unification of the vertices and of the edges be separate operations, other algorithms may unify vertices in constant time, then merge the edge sets as a separate linear-time step. It is this approach

which allows many of the algorithms implemented as graph operators to run in linear time, by working on the input edge set only, and, at the end, updating the edge set as per the disjoint sets for the next operator to be run.

The algorithm is relatively simple, and has four steps. First, the neighbor vertices in each neighbor list of $G.A$ are replaced by their representatives in $G.D$, and resulting self-loops are removed. Second, the neighbor list of each non-representative i is moved to the neighbor list of its representative $ki = G.D[i]$, with the neighbor lists of each non-representative left empty. Third, the adjacency list itself $G.A$ is shrunk to only contain neighbor lists for representatives, removing all the now empty neighbor lists for the non-representatives. Fourth and finally, any duplicate neighbors added to the neighbor lists are removed.

```
   Input: G
1  REPLACENEIGHBORSBYTHEIRREP()
2  MOVENONREPNEIGHBORSTOREPNEIGHBORS()
3  SHRINKADJACENCYLISTTOREPSONLY()
4  REMOVEDUPLICATENEIGHBORS()
```

Algorithm 16: UNIFYGRAPH

The functions involved in UNIFYGRAPH are implemented as follows.

```
   Input: G
1  Function REPLACENEIGHBORSBYTHEIRREP():
2      for 0 ≤ i < |G.A| do
3          ki ← G.D[i]
4          jy ← 0
5          for 0 ≤ jx < |G.A[i]| do
6              kj ← G.D[G.A[i][jx]]
7              if kj ≠ ki then
8                  G.A[i][jy] ← kj
9                  jy ← jy + 1
10             end
11         end
12         G.A[i].resize(jy)
13     end
14 Function MOVENONREPNEIGHBORSTOREPNEIGHBORS():
15     for 0 ≤ i < |G.A| do
16         ki ← G.D[i]
17         if ki ≠ i then
18             G.A[ki].push_back_all(G.A[i])
19             G.A[i].clear()
20         end
21     end
```

Algorithm 17: REPLACENEIGHBORSBYTHEIRREP, and MOVENONREPNEIGHBORSTOREPNEIGHBORS

The REPLACENEIGHBORSBYTHEIRREP algorithm modifies the adjacency list $G.A$, replacing each neighbor vertex by the representative of that vertex in $G.D$. The MOVENONREPNEIGHBORSTOREPNEIGHBORS algorithm modifies the adjacency list $G.A$ also, this time moving the edge sets of each non-representative neighbor into the edge set of their representative.

```
   Input: G
 1 Function SHRINKADJACENCYLISTTOREPSONLY():
 2     max ← 0
 3     for 0 ≤ i < |G.A| do
 4         ki ← G.D[i]
 5         if ki > max then
 6             
 7             max ← ki
 8         end
 9     end
10     G.A.resize(max + 1)
11 Function REMOVEDUPLICATENEIGHBORS():
12     found ← list of size |G.A| s.t. ∀0 ≤ i ≤ |G.A| : found[i] = ⊥
13     for 0 ≤ i < |G.A| do
14         jy ← 0
15         for 0 ≤ jx < |G.A[i]| do
16             k ← G.A[i][jx]
17             if found[k] ≠ i then
18                 found[k] ← i
19                 G.A[i][jy] ← k
20                 jy ← jy + 1
21             end
22         end
23         G.A[i].resize(jy)
24     end
```

Algorithm 18: SHRINKADJACENCYLISTTOREPSONLY, and REMOVEDUPLICATENEIGHBORS

The SHRINKADJACENCYLISTTOREPSONLY algorithm finds the representative in $G.D$ with the maximum unique identifier, then modifies $G.A$ to contain only edge lists for unique identifiers up to this maximum. In practice, all identifiers less than or equal to this maximum will necessarily be representatives. The REMOVEDUPLICATENEIGHBORS algorithm removes any duplicate neighbors added to the edge lists in $G.A$ by the applied functions, resizing each list to a contain only the representative neighbors.

Notably, the UNIFYGRAPH algorithm is performed in-place, with values of $G.A$ removed or replaced only, but not added, and without creating a copy of $G.A$. This achieves a linear time and space complexity, and in practice causes the transform to be highly efficient.

A2.4.2 ORDERBYLABELS

The ORDERBYLABELS algorithm gives an ordering over a list of strings according to their first letter, such that all strings with the same first letter are adjacent in the order. We use this algorithm in SORTGRAPHBYLABELS to order the graph with root vertices first, then sink vertices, then non-sink vertices, in $G.A$, $G.D$, and $G.L$.

```
   Input: strings, letters
   Output: o
 1 ∀0 ≤ i < |strings| : o[i] ← ⊥
 2 ∀0 ≤ i < |letters| : c[i] ← 0
 3 for 0 ≤ i < |strings| do
 4 |   for 0 ≤ j < |letters| do
 5 |   |   if strings[i][0] = letters[j] then
 6 |   |   |   o[i] ← c[j]
 7 |   |   |   c[j] ← c[j] + 1
 8 |   |   |   break
 9 |   |   end
10 |   end
11 end
12 for 1 ≤ i < |letters| do
13 |   c[i] ← c[i − 1] + 1
14 end
15 for 0 ≤ i < |strings| do
16 |   for 1 ≤ j < |letters| do
17 |   |   if strings[i][0] = letters[j] then
18 |   |   |   o[i] ← o[i] + c[j − 1]
19 |   |   |   break
20 |   |   end
21 |   end
22 end
23 return o
```

Algorithm 19: ORDERBYLABELS

The algorithm first initializes an empty mapping o, which will map from each index of a string in $strings$ to its new index in the resultant order. The list c is initialized also, which will hold a count for the number of strings in $strings$ having a first letter in $letters$, with one count per letter. Each string in the given list is iterated over, and the current count of the its first letter in c used as its index in o. The count for each letter is then replaced by the offset of the first string in the eventual order with that letter as its first. That is, s.t. each $c[i]$ is assigned $1 + c[i-1] + c[i-2] + \cdots + c[0]$. Finally, the algorithm adds the relevant offset in c to the previously computed index in $o[i]$, resulting in $o[i]$ mapping to a new unique index for the string $strings[i]$. The ordering o is then returned.

The algorithm is polynomial in general, with a runtime of $O(|strings| \times |letters|)$, however in practice it is only ever used with $letters = [\text{“}\texttt{r}\text{”}, \text{“}\texttt{h}\text{”}, \text{“}\texttt{v}\text{”}]$ and $strings = G.L$, hence under this binding is linear at $O(|G.L|)$.

A2.4.3 ORDERBYREPS

The ORDERBYREPS algorithm is similar to ORDERBYLABELS, however this time we return an order such that all disjoint set representatives occur in the order *before* any non-representative vertex identifiers. Any ordering between representatives is preserved, so that if representative u has a lesser unique identifier than representative v, it is placed at a lower index in the order. The same order-preservation property holds for non-representatives.

Input: G
Output: o
$o \leftarrow$ list of size $|G.ds|$ s.t. $\forall 0 \leq i < |G.ds| : o[i] = \perp$
$c \leftarrow 0$
for $i \in G.D$ **do**
  $o[i] \leftarrow c$
  $c \leftarrow c + 1$
end
for $0 \leq i < |G.ds|$ **do**
  if $o[i] = \perp$ **then**
    $o[i] \leftarrow c$
    $c \leftarrow c + 1$
  end
end
return o

Algorithm 20: ORDERBYREPS

The algorithm first iterates over each representative in $G.D$, in order of unique identifier, and assigns an index in the order o to an incrementing counter c. The algorithm then proceeds to loop over each remaining unique identifier in $G.ds$ that has not yet been assigned an index in o, and assigns an index again using the incrementing counter c. Finally, the order o is returned.

The algorithm is linear in runtime, performing a scan over $G.D$ and then $G.ds$, for an asymptotic bound of $O(|G.ds|)$ or equivalently $O(|G.L|)$.

A2.4.4 SORTGRAPH

The SORTGRAPH algorithm sorts $G.L$, $G.D$, and $G.A$ by a given order o. The ORDERBYLABELS and ORDERBYREPS both give such an order o, where the existing unique identifier

of each vertex u is mapped to a new unique identifier v, and $o[u] = v$. The runtime of this algorithm is linear at $O(|V| + |E|)$.

```
   Input: G, o
 1 Function SortByOrder(A):
 2 |   B ← Copy(A)
 3 |   for 0 ≤ i < |A| do
 4 |   |   A[o[i]] ← B[i]
 5 |   end
 6 Function ReplaceByOrder(A):
 7 |   for 0 ≤ i < |A| do
 8 |   |   A[i] = o[A[i]]
 9 |   end
10 SortByOrder(G.L)
11 SortByOrder(G.D)
12 SortByOrder(G.A)
13 ReplaceByOrder(G.D)
14 for 0 ≤ i < |G.A| do
15 |   ReplaceByOrder(G.A[i])
16 end
17 G.A.resize(|G.D|)
```

Algorithm 21: SORTGRAPH

The algorithm includes two nested function, SORTBYORDER, and REPLACEBYORDER. SORTBYORDER takes a list A, and moves the value at $A[i]$ to the new index given by the order o, such that $A[o[i]]$ gets the prior value of $A[i]$. REPLACEBYORDER also takes a list A, but this time assigns the value in the order at $o[A[i]]$ to $A[i]$. The vertex labels $G.L$, disjoint sets $G.D$, and adjacency list $G.A$ are first reordered by SORTBYORDER. The representatives for the disjoint sets $G.D$ are then assigned to be consistent with the new unique identifiers using a call to REPLACEBYORDER. Each neighbor list of the adjacency list $G.A$ is then updated with the new unique identifiers of the neighbors using REPLACEBYORDER. The adjacency list is then resized to the number of representatives $|G.D|$, so as to only contain neighbor lists of representative vertices.

Note that this assumes that the disjoint set representatives have lower unique identifiers than any non-representative. This will be true if the passed order is given by ORDERBYREPS, or if all vertices have singleton equivalence classes, which is a precondition of any algorithm calling ORDERBYLABELS. Hence the assumption will be satisfied if the passed order is generated by ORDERBYREPS or ORDERBYLABELS, which it is in all usage.

The runtime of SORTBYORDER and REPLACEBYORDER are linear on the given A. SORTBYORDER is called on $G.L$, $G.D$, and $G.A$; so the combined time complexity of these three calls is $O(|G.D| + |G.A| + |G.L|)$ which is $O(|G.L|)$, or equivalently, $O(|V|)$. The REPLACEBYORDER function is called once on $G.D$, then on each neighbor list of $G.A$. As the sum of the sizes of the neighbor lists is $|E|$, and $G.D$ is bounded by $|V|$, we have

$O(|V| + |E|)$ for all REPLACEBYORDER calls. Combining these two run times gives a total linear runtime for the algorithm of $O(|V| + |E|)$.

A2.4.5 SORTGRAPHBYREPS

The SORTGRAPHBYREPS algorithm calls UNIFYGRAPH(G) followed by SORTGRAPH(ORDERBYREPS(G), G). The result of any disjoint set unification via UNIFYSETS is applied to the graph, and the graph is then sorted to have all representatives before non-representatives in $G.L$, $G.D$, and $G.A$. As each of the algorithms these call are linear, so too is this algorithm itself, with a runtime of $O(|V| + |E|)$. This function is applied often, typically after the graph has been transformed by an operator, with vertices being merged into equivalence classes.

A2.4.6 SORTGRAPHBYLABELS

The SORTGRAPHBYLABELS algorithm simply calls SORTGRAPH(ORDERBYLABELS(G, [“r″, “h″, “v″]), G). The graph is sorted such that the root vertex is placed first (beginning with “r”), then the sink vertices (beginning with “h”), and then the non-sink vertices (beginning with “v”). This algorithm requires the precondition that all vertices are in their own equivalence class, and so is applied exactly once only after a graph is read from a file. We will see that this is done later on in the section on graph operator algorithms. The runtime of the algorithm is linear, as the single function it calls is linear, at a runtime of $O(|V| + |E|)$.

A3 Implementation of Algorithms

A3.1 Pre-processing

While thus far we have explored algorithms that pertain to the graph data structure, we have yet to show how the graph operators themselves are implemented. We must first introduce the notion of a graph's state, which captured as a series of boolean fields of G. A description of each field is as follows.

ifRootV: True if the graph has a root vertex, that is, a vertex with unique index 0 and a label beginning with “r”.
ifRootE: True if the graph has an edge with an endpoint at a root vertex.
ifAcyclic: True if the graph is acyclic.
ifBackwards: True if the edges of the graph are backwards directed, that is, from the sinks to the non-sinks as opposed to from the non-sinks to the sinks.
ifInitialized: True if the graph has been initialized, that is, populated from a file.
ifSinkReachable: True if all vertices in the graph reach a sink if the graph is forwards directed, or, if the graph is backwards directed, true if every vertex may be reached from a sink.
ifSinks: True if there are sink vertices in the graph.

The state of the graph is checked as a pre-condition before each graph operation is applied, with each operator implementation defining its own precondition as an assertion over the state of the graph. Similarly, the graph state may modified by an operator application, and so each operator implementation includes a post-condition which will hold over the graph state after application of that operator. A set of common functions are used to transform a graph prior to assertion of the pre-condition. These are known as pre-processing algorithms, and also have pre and post conditions.

The difference between a pre-processing algorithm and an operator algorithm is that pre-processing algorithms are not made available to the user as operator via the CLI. Additionally, operator algorithms do not call other operator algorithms in their main algorithm, but operator algorithms may call pre-processing algorithms and other operator algorithms in the pre-processing step, prior to assertion of their pre-condition. As an example, a pre-processing algorithm may reverse the edges of a forwards directed graph prior to application of an operator requiring a backwards directed graph as input. All pre-processing algorithms are trivially linear in runtime, as either $O(|V|)$, $O(|E|)$, or $O(|V| + |E|)$. We now continue to the set of pre-processing algorithms.

A3.1.1 ADDEDGESFROMROOTTOSINKS

The ADDEDGESFROMROOTTOSINKS pre-processing algorithm adds an edge from the root vertex to all sink vertices.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	T	_	T	T	_	T

The pre-condition table above shows that it requires a root vertex to exist (ifRootV), sink vertices to exist (ifSinks), the edges of the graph to be backwards directed (ifBackwardsDirected), and to have been initialized (ifInitialized). We see that the ifAcyclic and ifSinkReachable fields may be any value, as indicated with −.

```
1 for u ∈ G.D do
2 |   if u ∈ G.S then
3 |   |   G.A[0].push_back(u)
4 |   end
5 end
```

Algorithm 22: ADDEDGESFROMROOTTOSINKS

The algorithm itself is trivial, iterating over each representative u in $G.D$, checking if that representative is a sink with $u \in G.S$, and adding that vertex to the neighbor set of the root at $G.A[0]$.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
T	T	_	T	T	_	T

The post-condition table above shows the state of graph after application of the algorithm. All fields remain the same as in the pre-condition table, except for the ifRootE field, indicated in bold. This field has been changed from false to true, as per the addition of root edges to the graph.

A3.1.2 ADDEDGESFROMSINKSTOROOT

The ADDEDGESFROMSINKSTOROOT pre-processing algorithm performs the inverse of ADDEDGESFROMROOTTOSINKS, by adding an edge from each sink to the root in a forwards directed graph.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	T	_	F	T	_	T

The pre-condition table is the same as ADDEDGESFROMROOTTOSINKS, except that this time the graph is required to be forwards directed by the value true of ifBackwards.

```
1 for u ∈ G.D do
2 |   if u ∈ G.S then
3 |   |   G.A[u].push_back(0)
4 |   end
5 end
```

Algorithm 23: ADDEDGESFROMSINKSTOROOT

The algorithm iterates over each representative u, checks if it is a sink, and, if so, adds 0 (the root vertex) to the neighbor list of u.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
T	T	_	F	T	_	T

The post-condition again changes only the field of ifRootE from false to true.

A3.1.3 REMOVEEDGESFROMROOTTOSINKS

The REMOVEEDGESFROMROOTTOSINKS pre-processing algorithm removes all edges from root vertices to sink vertices in a backwards directed graph.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
T	T	_	T	T	_	T

The precondition table is the same as the post-condition table ADDEDGESFROMROOTTO-SINKS.

```
1 G.A[0].clear()
```

Algorithm 24: REMOVEEDGESFROMROOTTOSINKS

The algorithm simply clears the edge list of the root vertex 0 by executing $G.A[0]$.clear().

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	T	_	T	T	_	T

The post-condition table is then the same as the pre-condition table, except that the ifRootE field is changed from true to false.

A3.1.4 REMOVEEDGESFROMSINKSTOROOT

The REMOVEEDGESFROMSINKSTOROOT pre-processing algorithm removes all edges from sink vertices to the root vertex in a forwards directed graph.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
T	T	_	F	T	_	T

The precondition table is the same as the post-condition table ADDEDGESFROMSINKSTOROOT.

```
1 for u ∈ G.D do
2 |   if u ∈ G.S then
3 |   |   G.A[u].clear()
4 |   end
5 end
```

Algorithm 25: REMOVEEDGESFROMSINKSTOROOT

The algorithm iterates over each representative, checks if it is a sink, and, if so, clears its edge list.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	T	_	F	T	_	T

Again, the post-condition table is the same as the pre-condition table but for the field ifRootE, which was true and now is false.

A3.1.5 REVERSEEDGES

The REVERSEEDGES pre-processing algorithm reverses the edges of the graph, such that a forwards directed graph becomes backwards directed, and vise-versa.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
_	_	_	_	T	_	_

The pre-condition requires only that the graph is initialized. The algorithm calls the TRANSPOSE function defined on the adjacency list in Algorithm 10.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
_	_	_	$*$	T	_	_

The post-condition updates the value of ifBackwards, changing it to true if false and false if true, as indicated by the $*$.

A3.1.6 ADDROOTVERTEX

The ADDROOTVERTEX algorithm adds a root vertex to the graph with a unique id of 0 and a label of "`r`".

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	**F**	_	_	T	_	T

The pre-condition requires that the graph have no existing root vertex (ifRootV) or edges (ifRootE), has sink vertices (ifSinks), and is initialized (ifIntialized).

1 $G.L$.push_back("`r`")
2 $G.A$.resize($|G.L|$)
3 $G.D$.push_back($|G.D|$)
4 SORTGRAPHBYLABELS(G)

Algorithm 26: ADDROOTVERTEX

The algorithm adds a new label "`r`" to $G.L$, resizes $G.A$ to $G.L$'s new size, and adds a new singleton disjoint set to $G.D$ for the root. The algorithm then calls SORTGRAPHBYLABELS to re-order the graph with the root vertex at unique identifier 0, followed by the sinks at greater unique identifiers, then the non-sinks.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	**T**	_	_	T	_	T

The post-condition table changes (ifRootV) from false to true as compared to the pre-condition table.

A3.1.7 TURNLEAVESINTOSINKS

The TURNLEAVESINTOSINKS algorithm transforms all vertices in the graph which have no (outbound) neighbors into sink vertices.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
_	_	_	_	T	_	_

The pre-conditions require only that the graph be initialized.

```
1 for u ∈ G.D do
2 |   if |G.A[u]| = 0 then
3 |   |   G.L[u] ← "h_" + G.L[u]
4 |   end
5 end
```

Algorithm 27: TURNLEAVESINTOSINKS

The algorithm iterates over each representative u, then checks if the size of the neighbor list of u is 0 in $G.A$. Vertices with such empty lists are called a leaf. If u is a leaf, then the label u is pre-pended with "h_", signifying it as a sink vertex, and rendering the expression $u \in G.S$ now true.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
_	_	_	_	T	**T**	**T**

The post-condition table changes ifSinks to true, as either at least one new sink has been added or all vertices are involved in a cycle. The software assumes that this is never the case globally, as a totally cyclic graph has a reachability relation which is true on every pair of vertices, and use of this software on such a graph would serve no purpose. The post-condition table also asserts the ifSinkReachable field, as all sinks that have been added to the graph must be reached by some other non-sink, otherwise that non-sink would itself be a leaf, and thus transformed into a sink by application of the algorithm.

A3.2 Basic Operators

We are now ready to present the algorithms for the operators themselves. Each operator is given with the letter that is used for it in the CLI – in the arguments to the `-algorithms`, `-block`, `-prefix`, and `-suffix` options. The operators have an optional pre-processing step in which they may call pre-processing algorithms, then a table of pre-conditions asserted over the graph state, followed by the algorithm implementing the operation, and finally the post-condition table.

A3.2.1 OPERATOR O

The first operator we will introduce is **O**, implemented via the OPERATOR O algorithm. It reads a graph from a file, initializing the disjoint sets $G.D$, the vertex labels $G.L$, and the adjacency list $G.A$.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	F	F	F	**F**	F	F

The preconditions here require that all graph state fields be false, as no operator or pre-processing algorithm has yet been applied on them to assert a post-condition. The algorithm

reads an input file, line by line. Each line is either a vertex label, or two vertex labels separated by a space, denoting an edge. The labels and edges are propagated to $G.L$, $G.A$, $G.D$.

Input: $G, file$

```
1  M ← map from vertex labels to unique identifiers
2  Function GetId(k):
3      if k ∉ G.V and k ∉ G.S and k ∉ G.R then
4          k ← "v_" + k
5      end
6      if M[k] = ⊥ then
7          M[k] ← |M|
8          G.L.push_back(k)
9          G.A.push_back([])
10     end
11 for line ∈ file do
12     (l, r) ← line.split(" ")
13     u ← GetId(l)
14     if r ≠ " " then
15         v ← GetId(r)
16         if v ∉ G.A[u] then
17             G.A[u].push_back(v)
18         end
19     end
20 end
21 G.D ← disjoint sets s.t. ∀0 ≤ i < |G.A| : G.ds[i] ← i
```

Algorithm 28: OPERATOR O

The algorithm takes an empty graph G and a file. The algorithm starts by defining a nested function called GETID, which, given some vertex label k, returns the unique identifier for k in the map M. Within GETID, on line 3, the algorithm first checks if the given label k is a root, sink, or normal vertex – if it is none of these, then it is made a normal vertex by pre-pending "v_" to k. Then, if k is not yet in the map M, the size of M is assigned to $M[k]$, giving $|M|$ as the unique identifier for k. Next, k is appended to $G.L$, and an empty neighbor list appended to $G.A$.

Outside the definition of GETID, the algorithm begins reading the graph by iterating over each line of a file. The line is split on the space character into a left component l, and a right component r. Then, u is assigned to the call of GETID on l, and, if r is an empty string, v assigned to the call of GETID on r. Also, if v is not already in the neighbor list of u, it is added to that neighbor list. After the file has been read, the disjoint sets are initialized with each vertex having itself as its own representative – that is, $G.D[i] = i$ for all i.

The runtime of the algorithm is linear $O(|V| + |E|)$, assuming that M uses a constant time hash function and there are no duplicate edges present.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	F	F	F	**T**	F	_

The post condition states that the graph is now initialized (isInitialized is true), and whether there were any sinks encountered – that is, labels beginning with "h".

A3.2.2 OPERATOR C

The **C** operator adds a root vertex with an edge to each sink, and leaves the graph in a backwards directed state. This is typically called as an argument to the `--prefix` CLI option, as the root vertex allows for processing by many other operators (e.g. D, F, N, etc.).

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	F	F	F	T	F	T

The algorithm calls ADDROOTVERTEX, REVERSEEDGES, and then ADDEDGESFROMROOTTOSINKS.

As the implementation **C** calls three other linear-time algorithms, the runtime of **C** is, trivially, linear.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
T	T	F	T	T	F	T

The post-condition reflects the addition of the root vertex, and edges, with true values for ifRootE and ifRootV. As the edges are backwards directed, ifBackwardsDirected is true also.

A3.2.3 OPERATOR S

The **S** operator applies the SCC reduction on the graph. This is the first operator algorithm we encounter with a pre-processing step, given below.

$$\neg G.\text{ifRootV} \rightarrow \text{ADDROOTVERTEX}(G)$$
$$\neg G.\text{ifBackwards} \rightarrow \text{REVERSEEDGES}(G)$$
$$\neg G.\text{ifRootE} \rightarrow \text{ADDEDGESFROMROOTTOSINKS}(G)$$

A pre-processing step checks the state of the graph, and applies pre-processing algorithms such that the pre-condition of the operator algorithm holds. Here, if the graph does not have a root vertex ($\neg G.ifRootV$), we call ADDROOTVERTEX to add the root vertex. Similarly, if the graph is not backwards directed, then REVERSEEDGES is called, and if the graph does not have root edges, then ADDEDGESFROMROOTTOSINKS is called.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
T	T	**F**	T	T	_	T

The pre-conditions in the table above check that the graph has a root vertex, is backwards directed, and has edges – these conditions are ensured to be true by the pre-processing step. The pre-condition asserts that ifAcyclic is false, as applying **S** on an acyclic graph has no effect, and so should not occur. The pre-conditions also check if the graph is initialized, and has sinks.

The operator uses Tarjan's SCC algorithm [75] on the adjacency list $G.A$,and a designated root vertex of 0, to unify disjoint sets in $G.D$. We have described the procedure in the body of this thesis in Algorithm 5. After the SCC reduction is run, $G.D$ has been modified with unifications corresponding to the strongly-connected components in $S(G)$. In order to unify the associated edge sets, SORTGRAPHBYREPS is called on the transformed G, to apply the unifications to the graph.

The **S** operator has linear runtime by the linearity in runtime of Tarjan's algorithm, and also SORTGRAPHBYREPS.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
T	T	**T**	T	T	_	T

The post-condition makes only one update to the graph, changing ifAcyclic from true to false, as the graph is now acyclic.

A3.2.4 OPERATOR T

The operator **T** removes all non-sink vertices from the graph that do not reach a sink.

$$\neg G.\text{ifAcyclic} \rightarrow \text{OPERATOR S}(G)$$
$$\neg G.\text{ifRootV} \rightarrow \text{ADDROOTVERTEX}(G)$$
$$\neg G.\text{ifBackwards} \rightarrow \text{REVERSEEDGES}(G)$$
$$\neg G.\text{ifRootE} \rightarrow \text{ADDEDGESFROMROOTTOSINKS}(G)$$

The pre-processing stage ensures that the graph is acyclic (using OPERATOR S), that edges are backwards directed, and that a root vertex exists with edges to each sink.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
T	T	T	T	T	F	T

The pre-conditions here are the same as the post-conditions of OPERATOR S, however this time with ifSinkReachable explicitly required to be false. Because this algorithm removes all non-sink reachable vertices, having this precondition as false ensures the algorithm is not performed more than once, as this would be redundant.

The algorithm works by conducting a DFS from the root, through all sinks, to all of the non-sinks they reach in the backwards directed graph. Any non-sinks not reached by a sink in the backwards graph must necessarily not reach a sink in the forwards directed graph, and so may be deleted.

```
   Input: G, r
 1 o ← TOPOLOGICALORDER(G, r)
 2 for 0 ≤ i < |G.A| do
 3 |   if o[i] = ⊥ then
 4 |   |   G.UNIFYSETS(r, i)
 5 |   |   G.A[i].clear()
 6 |   else
 7 |   |   jy ← 0
 8 |   |   for 0 ≤ jx < |G.A| do
 9 |   |   |   k ← G.A[i][jx]
10 |   |   |   if o[k] ≠ ⊥ then
11 |   |   |   |   G.A[i][jy] = k
12 |   |   |   |   jy ← jy + 1
13 |   |   |   end
14 |   |   end
15 |   |   G.A[i].resize(jy)
16 |   end
17 end
18 SORTGRAPHBYREPS(G)
```

Algorithm 29: OPERATOR T

The algorithm takes a graph G and the root vertex r of G. Initially, a topological order is computed from r as o. Where $o[i] = \perp$, the vertex with index i has not been visited in computation of the topological order, and so must not not be reachable from the root. As the root is connected to all sinks, any vertex that is not reached from the root must not be reachable from a sink.

On line 2, the algorithm iterates over each vertex of the graph. If that vertex is unvisited in the topsort, i.e. if $o[i] = \perp$, then it is unified with the root vertex r, and its list of edges cleared in the adjacency list A. Otherwise, another loop re-constructs the edge list of i in $G.A$, such that $G.A[i]$ contains no vertices unreachable vertices k such that $o[k] = \perp$.

Finding all unvisited vertices is a DFS search, and so the runtime of this component is $O(|V| + |E|)$. The re-ordering of the graph is also $O(|V| + |E|)$, as each vertex and edge is visited no more than once – the resize operation takes constant time, as no more memory is allocated to an edge list, only the last members deleted. Hence, the runtime of OPERATOR T is linear at $O(|V| + |E|)$.

The runtime of OPERATOR T is linear at $O(|V| + |E|)$.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
T	T	T	T	T	T	T

The post-conditions here are the same as the pre-conditions, except with ifSinkReachable changed to true, as all sinks are now reachable from the root as per the operation performed here.

A3.3 Sink-reachability Preserving Compression Operators

A3.3.1 OPERATOR D

The next operator, **D**, implements the dominator reduction.

$$\neg G.\text{ifAcyclic} \rightarrow \textsc{Operator S}(G)$$
$$\neg G.\text{ifSinkReachable} \rightarrow \textsc{Operator T}(G)$$
$$\neg G.\text{ifRootV} \rightarrow \textsc{AddRootVertex}(G)$$
$$G.\text{ifBackwards} \rightarrow \textsc{ReverseEdges}(G)$$
$$\neg G.\text{ifRootE} \rightarrow \textsc{RemoveEdgesFromSinksToRoot}(G)$$

The pre-processing establishes an acyclic, forwards directed graph, with a root vertices and root edges, as the input to the algorithm. The operators OPERATOR S and OPERATOR T are called if the post-conditions they establish are not yet true of the graph (specifically, ifAcyclic and ifSinkReachable). The pre-processing algorithms ADDROOTVERTEX, REVERSEEDGES, and REMOVEEDGESFROMSINKSTOROOT are then used to ensure the existence a root vertex, reverse the edges, if the graph is not already forwards directed, and then, add edges between all roots and sinks, if they do not already exist.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
T	T	T	F	T	T	T

This uses the dominator algorithm given in the body of this thesis (see Algorithm 6) to unify vertices in $G.D$ according to the dominator compression. As with SCC, SORTGRAPHBYREPS is called to propagate the unifications made in $G.D$ on the vertices, to the corresponding unifications of edge sets in $G.A$.

The algorithm is linear in time complexity, as all pre-processing algorithms are linear time, the SORTGRAPHBYREPS algorithm is linear time, and the dominator algorithm is linear by the proof in [19].

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
T	T	T	F	T	T	T

A3.3.2 OPERATOR F

F implements the M_i successor-based modular decomposition.

$$\neg G.\text{ifAcyclic} \rightarrow \text{OPERATOR S}(G)$$
$$\neg G.\text{ifSinkReachable} \rightarrow \text{OPERATOR T}(G)$$
$$\neg G.\text{ifRootV} \rightarrow \text{ADDROOTVERTEX}(G)$$
$$\neg G.\text{ifBackwards} \rightarrow \text{REVERSEEDGES}(G)$$
$$G.\text{ifRootE} \rightarrow \text{REMOVEEDGESFROMSINKSTOROOT}(G)$$

The pre-processing ensures that the graph is acyclic, all non-sinks reach sinks, there is a root vertex, the graph is backwards directed, and, there are no root edges.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	T	T	T	T	T	T

The pre-conditions assert the conditions ensured by the pre-processing stage, assume that the graph has sinks, and is initialized.

This operator uses Algorithm 7 presented in the body of this thesis, then calls SORT-GRAPHBYREPS to unify the edges in $G.A$ in the same manner as **S** and **D**. The algorithm is linear in time complexity, as all pre-processing algorithms are linear time, the SORT-GRAPHBYREPS algorithm is linear time, and the M_i modular decomposition is linear by the proof in [19].

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	T	T	T	T	T	T

The post-conditions here are the same as the pre-conditions.

A3.3.3 OPERATOR N

The **N** operator implements the M_c successor-based modular decomposition.

$$\neg G.\text{ifAcyclic} \rightarrow \text{OPERATOR S}(G)$$
$$\neg G.\text{ifSinkReachable} \rightarrow \text{OPERATOR T}(G)$$
$$\neg G.\text{ifRootV} \rightarrow \text{ADDROOTVERTEX}(G)$$
$$G.\text{ifBackwards} \rightarrow \text{REVERSEEDGES}(G)$$
$$G.\text{ifRootE} \rightarrow \text{REMOVEEDGESFROMSINKSTOROOT}(G)$$

The pre-processing ensures an ayclic, sink-reachable, rooted graph, with all edges forwards directed, and no root edges.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	T	T	F	T	T	T

The pre-conditions assert the graph is initialized and has sinks, as well as all the conditions already ensured on the graph state by the pre-processing.

The **N** operator uses Algorithm 8 presented in the body of this thesis, calling SORTGRAPHBYREPS like in **S**, **D**, and **F**. The algorithm is linear in time complexity, as all pre-processing algorithms are linear time, the SORTGRAPHBYREPS algorithm is linear time, and the M_c modular decomposition is linear by the proof in [19].

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	T	T	F	T	T	T

The post-conditions are unchanged from the pre-conditions.

A3.4 General Reachability Preserving Compression Operators

A3.4.1 OPERATOR B

Operator **B** implements the bottom-up transitive reduction from [87].

$$\neg G.\text{ifAcyclic} \rightarrow \text{OPERATOR S}(G)$$
$$\neg G.\text{ifSinkReachable} \rightarrow \text{OPERATOR T}(G)$$
$$\neg G.\text{ifRootV} \rightarrow \text{ADDROOTVERTEX}(G)$$
$$G.\text{ifBackwards} \rightarrow \text{REVERSEEDGES}(G)$$
$$G.\text{ifRootE} \rightarrow \text{REMOVEEDGESFROMSINKSTOROOT}(G)$$

The pre-conditions here are the same as for operator **N**.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	T	T	F	T	T	T

This uses code directly provided by the authors of [87]. An adapter is created over the interface of that code, allowing the operator implementation to construct the graph for the bottom-up transitive reduction by passing an instance of the graph data structure G. The bottom-up transitive reduction algorithm may then be called, and the edges in G updated by the according transformation. Note that it is not necessary to call SORTGRAPHBYREPS after this operation, as the bottom-up transitive reduction removes edges only, and does not modify the equivalence classes of the graph vertices (i.e. the representatives).

While the algorithm has worst case time complexity of $O(|V|^3)$, heuristics may be employed that give much better results in practice [87].

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	T	T	F	T	T	T

The post-condition is the same as the precondition.

A3.4.2 OPERATOR L

The **L** operator implements the linear-equivalence relation reduction as introduced in [22].

$$\neg G.\text{ifAcyclic} \rightarrow \text{OPERATOR S}(G)$$
$$\neg G.\text{ifSinkReachable} \rightarrow \text{OPERATOR T}(G)$$
$$\neg G.\text{ifRootV} \rightarrow \text{ADDROOTVERTEX}(G)$$
$$G.\text{ifBackwards} \rightarrow \text{REVERSEEDGES}(G)$$
$$G.\text{ifRootE} \rightarrow \text{REMOVEEDGESFROMSINKSTOROOT}(G)$$

The pre-processing and preconditions are the same as **B** and **N**.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	T	T	F	T	T	T

The implementation also uses the same code as **B**, with an class used to construct the graph by passing G. The linear equivalence relation algorithm is called, and the disjoint sets $G.D$ updated via the computed equivalence classes. SORTGRAPHBYREPS is then called on the graph G, to merge the new equivalence classes.

The algorithm has high time complexity at $O(|V| \times (|V| + |E|))$, as well as high space complexity, at $O(|V|^2)$ – however, by applying the transitive reduction initially, the complexity of computing the linear equivalence relation is reduced to a time of $O(|V| + |E|)$ and a space of $O(|V|)$ on the transformed graph [22].

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	T	T	F	T	T	T

The post-conditions, again, are the same as the pre-conditions.

A3.5 Reachability Index Operators

A3.5.1 OPERATOR I

The **I** operator computes the reachability index via the two-hop label. The two-hop label, introduced in the work of Cohen *et al.* [15], computes two functions $L.in$ and $L.out$ given a graph G, such that for vertices u and v of G, if $L.in(u) \cap L.out(v) \neq \varnothing$, then u reaches v in G. $L.in$ and $L.out$ are implemented as adjacency lists, and so the space complexity is $O(|V|^2)$ in the worst case. However, as $|L.in(u)|$ and $|L.out(u)|$ for each vertex u are typically small in practice, the space is much smaller. Accessing the $L.in$ or $L.out$ set is a constant time operation, while comparing their sets is $O(|nlogn|)$ where $n = \max(|L.in(u)|, |L.out(v)|)$ for some u, v. Again, as n is typically small, the lookup is usually fast in practice.

$$\neg G.\text{ifAcyclic} \rightarrow \textsc{Operator S}(G)$$
$$\neg G.\text{ifSinkReachable} \rightarrow \textsc{Operator T}(G)$$
$$\neg G.\text{ifRootV} \rightarrow \textsc{AddRootVertex}(G)$$
$$G.\text{ifBackwards} \rightarrow \textsc{ReverseEdges}(G)$$
$$G.\text{ifRootE} \rightarrow \textsc{RemoveEdgesFromSinksToRoot}(G)$$

The pre-processing and preconditions here are the same as **B**, **N**, and **L**, and **V**.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	T	T	F	T	T	T

The implementation of the algorithm used to compute the label is from the same codebase as used for **B** and **L**, obtained with the authors' permission, and used via a simple adapter class to pass the graph G. Again, we omit this algorithm as it is not part of our original work, only its use and integration within the framework is part of our contribution.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	T	T	F	T	T	T

The post-conditions are the same as the pre-conditions.

A3.5.2 OPERATOR K

The OPERATOR Kalgorithm obtains the size of the two-hop label computed by **I**.

$$\neg G.\text{ifAcyclic} \rightarrow \textsc{Operator S}(G)$$
$$\neg G.\text{ifSinkReachable} \rightarrow \textsc{Operator T}(G)$$
$$\neg G.\text{ifRootV} \rightarrow \textsc{AddRootVertex}(G)$$
$$G.\text{ifBackwards} \rightarrow \textsc{ReverseEdges}(G)$$
$$G.\text{ifRootE} \rightarrow \textsc{RemoveEdgesFromSinksToRoot}(G)$$

The pre-processing and preconditions here are the same as **I**, as this operator merely counts the label constructed by **I**.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	T	T	F	T	T	T

The **K** operator uses the implementation of the two-hop label in **I**. The size of the two-hop label is computed as the sum of the sizes of each nested sub-list in $L.in$ and $L.out$.

A3.6 Utility Graph Operators

A3.6.1 OPERATOR X

The **X** operator constructs the mirror graph to answer aliasing queries typical of a points-to analysis, such as whether two non-sink vertices u and v share at least one sink vertex, i.e., whether $r(u) \cap r(v) \neq \emptyset$.

$$\neg G.\text{ifAcyclic} \rightarrow \text{OPERATOR S}(G)$$
$$\neg G.\text{ifSinkReachable} \rightarrow \text{OPERATOR T}(G)$$
$$\neg G.\text{ifRootV} \rightarrow \text{ADDROOTVERTEX}(G)$$
$$G.\text{ifBackwards} \rightarrow \text{REVERSEEDGES}(G)$$
$$G.\text{ifRootE} \rightarrow \text{REMOVEEDGESFROMSINKSTOROOT}(G)$$

The pre-processing and preconditions are the same as **I** and **K**.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	T	T	F	T	T	T

The basic procedure is described as follows. A copy of the graph is made, with all edges reversed. The sink vertices of the original and the copy are merged. The existing sink vertices are transformed into non-sink vertices, and the non-sink vertices in the copied half of the graph are transformed into sinks. The sink-reachability of a vertex u in the mirror graph is the set of all vertices v that share a reachable sink with u in the original graph.

Input: G
1 $n = |G.L|$
2 $i \leftarrow 1$
3 **while** $G.L[i] \in G.S$ **do**
4 $\quad i \leftarrow i + 1$
5 **end**
6 MIRRORLABELS(G, n)
7 MIRRORDISJOINTSETS(G, n)
8 MIRRORADJACENCYLIST(G, n)
9 $o \leftarrow$ GETMIRRORORDER(G, n, i)
10 SORTGRAPH(G, o)
11 SORTGRAPHBYREPS(G)
12 UNIFYMIRROREDSINKS(G)
13 SWITCHLABELSWITHUNDERSCORE(G)

Algorithm 30: OPERATOR X

The algorithm is split into multiple functions, each called by the top-level algorithm for the operator as given below. The algorithm first obtains n as the current size of $G.L$ – i.e. the number of vertices of the graph, whether they are representatives or not – and i as the maximum unique identifier of a sink. The mirror construction then proceeds over $G.L$,

$G.D$, and $G.A$. A mirror vertex is constructed for each existing unique identifier – whether representative or not – in the graph. For an existing unique identifier u, the new unique identifier in the mirror is $u + n - 1$.

The MIRRORLABELS function resizes the labels $G.L$, and computes a new label for each identifier as the concatenation of the old identifier with an underscore and then the old identifier repeated. The MIRRORDISJOINTSETS function resizes the $G.ds$ array backing $G.D$, then assigns representatives for the new identifier equivalent to the mirrored identifier of their representative. The adjacency list is then mirrored in MIRRORADJACENCYLIST, with backwards directed edges added for each pair of mirrored identifiers, whose corresponding old identifiers had a forwards directed edge. The GETMIRRORORDER function gives an ordering o on the unique identifiers of G, where $o[u]$ is the index of a unique identifier in the ordering. The ordering of the unique identifiers accords to the following groups.

(1) Sink typed, not mirrored, disjoint set representative.
(2) Sink typed, mirrored, disjoint set representative.
(3) Not sink typed, not mirrored, disjoint set representative.
(4) Not sink typed, mirrored, disjoint set representative.
(5) Not mirrored, not disjoint set representative.
(6) Mirrored, not disjoint set representative.

The ordering o is applied on the graph via a call to SORTGRAPH, defined in Subsection A2.4 on graph mutation algorithms. Next, the non-mirrored sinks are unified with the mirrored sinks in UNIFYMIRROREDSINKS, and SORTGRAPHBYREPS is called to apply the unification on the graph as a whole. Finally, the mirrored sinks have their label transformed to become non-sinks, while the mirrored non-sinks become sinks, using SWITCHLABELSWITHUNDERSCORE.

We now provide more detail on each of the subroutines involved in the mirror algorithm.

Function MIRRORLABELS(G, n)**:**
  $G.L$.resize$(2 \times n - 1)$
  for $1 \leq i < n$ **do**
    $G.L[n + i - 1] = G.L[i] +$ "_" $+ G.L[i]$
  end
Function MIRRORDISJOINTSETS(G, n)**:**
  $G.ds$.resize$(2 \times n - 1)$
  for $1 \leq i < n$ **do**
    $G.ds[n + i - 1] \leftarrow G.ds[i] + n - 1$
  end
Function MIRRORADJACENCYLIST(G, n)**:**
  $G.A$.resize(n)
  $B \leftarrow G.A$.copy$()$
  B.transpose$()$
  for $1 \leq i < |B|$ **do**
    for $0 < j < |B[i]|$ **do**
      $G.A[n + i - 1][j] \leftarrow B[i][j] + n - 1$
    end
  end

Algorithm 31: MIRRORLABELS, MIRRORDISJOINTSETS, and MIRRORADJACENCYLIST

This first set of functions gives the MIRRORLABELS, MIRRORDISJOINTSETS, and MIRRORADJACENCYLIST algorithms; which updates $G.L$, $G.D$, and $G.A$ with the unique identifiers, representatives, labels, and edges of the mirror graph.

Function GETMIRRORORDER(G, n)**:**
  $o \leftarrow$ list of size $2n-1$ s.t. $\forall 0 \leq i < 2n-1 : is[i] = \perp$
  $o[0] \leftarrow 0$
  $i \leftarrow 1$
  while $i < |o|$ ***and*** $G.L[i] \in G.S$ ***and*** $G.D[i] = i$ **do**
    $o[i] \leftarrow i$
    $i \leftarrow i + 1$
  end
  $k \leftarrow i$
  $j \leftarrow n$
  while $i < |o|$ ***and*** $G.L[j] \in G.S$ ***and*** $G.D[j] = j$ **do**
    $o[j] \leftarrow i$
    $i \leftarrow i + 1$
    $j \leftarrow j + 1$
  end
  while $i < |o|$ ***and*** $G.L[k] \in G.V$ ***and*** $G.D[k] = k$ **do**
    $o[k] \leftarrow i$
    $i \leftarrow i + 1$
    $k \leftarrow k + 1$
  end
  while $i < |o|$ ***and*** $G.L[j] \in G.V$ ***and*** $G.D[j] = j$ **do**
    $o[j] \leftarrow i$
    $i \leftarrow i + 1$
    $j \leftarrow j + 1$
  end
  while $k < n$ **do**
    $o[k] = i$
    $i \leftarrow i + 1$
    $k \leftarrow k + 1$
  end
  while $j < (2n-1)$ **do**
    $o[j] = i$
    $i \leftarrow i + 1$
    $j \leftarrow j + 1$
  end
  return o

Algorithm 32: GETMIRRORORDER

The GETMIRRORORDER function obtains the ordering with representatives placed before non-representatives, sinks before non-sinks, and non-mirrored vertices before mirrored vertices (in that order of precedence).

```
1  Function UNIFYMIRROREDSINKS(G):
2      j ← 1
3      while G.L[i] ∈ G.S do
4          G.D.UNIFYSETS(j, i)
5          i ← i + 1
6          j ← j + 1
7      end
8  Function SWITCHLABELSWITHUNDERSCORE(G):
9      for l ∈ G.L do
10         if HASUNDERSCORE(l) then
11             if l ∈ G.V then
12                 l ← "h_" + l
13         end
14         else
15             if l ∈ G.S then
16                 l ← "v_" + l
17             end
18         end
19     end
```

Algorithm 33: UNIFYMIRROREDSINKS, and SWITCHLABELSWITHUNDERSCORE

The third set of functions begins with UNIFYMIRROREDSINKS, which iterates over each mirrored sink representative (which by this call begins at the passed index i), and unifies it with each mirrored sink representative (which now begin at the index 1, and have a final index of the given $i - 1$).

Finally, SWITCHLABELSWITHUNDERSCORE iterates over all vertex labels of $G.L$, and if the label has an underscore character (a "_"), then we assume it is a mirrored vertex as per the label reassignment in the earlier call to MIRRORLABELS. If a mirrored label is a non-sink, it becomes a sink by pre-pending "h_" to the label, while if it is a sink, then "v_" is pre-pended to the label to make it a non-sink.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	T	T	F	T	T	T

The post-conditions are the same as the preconditions.

A3.6.2 OPERATOR V

The **V** operator computes kernel condensation as described in Section 4.3. The kernel condensation is of G is the graph with the minimum number of disjoint set representatives

such that all vertices have the same sink-reachability. This may be equivalently stated as the best possible sink-reachability reduction that can be obtained of G. As a proof of the optimality of our kernel condensation algorithm is left to future work, our implementation obtains a *close approximation* to the best possible kernel.

Additionally, as computing kernel condensation requires at least quadratic time in the worst case, it is impractical to do this for large graphs. This evidences further why the linear time sink-reachability preserving compression operators are necessary. The kernel is useful in comparing the compression achieved by sink-reachability preserving compression operators, or sequences thereof, to a close approximation of an best-case compression.

$$\begin{gathered}\neg G.\text{ifAcyclic} \rightarrow \textsc{Operator S}(G)\\ \neg G.\text{ifSinkReachable} \rightarrow \textsc{Operator T}(G)\\ \neg G.\text{ifRootV} \rightarrow \textsc{AddRootVertex}(G)\\ G.\text{ifBackwards} \rightarrow \textsc{ReverseEdges}(G)\\ G.\text{ifRootE} \rightarrow \textsc{RemoveEdgesFromSinksToRoot}(G)\end{gathered}$$

The pre-processing and preconditions here are the same as **B**, **N**, and **L**.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	T	T	F	T	T	T

The algorithm computes the vertex kernel, obtained by performing a DFS search to obtain all pairs of non-sink to sink reachability pairs, constructing a bipartite graph with these pairs as edges, merging vertices that reach only one sink with the sink they reach, and merging all non-sinks that reach the same set of sinks together.

Input: G
1 $M \leftarrow$ adjacency list of size $|G.A|$
2 $M \leftarrow \textsc{GetBipartideSinkReachabilityGraph}(G)$
3 **if** $\textsc{HasAnyEdges}(M)$ **then**
4 $\quad \textsc{MergeSingleSinkReachingVerticesWithSink}(G, M)$
5 $\quad o \leftarrow \textsc{GetNeighborListOrder}(G, M)$
6 $\quad i \leftarrow \textsc{GetIndexOfFirstNonEmptyNeighborList}(G, M, o)$
7 $\quad \textsc{MergeVerticesWithSameSinkReachability}(G, M, i)$
8 **end**
9 $\textsc{SortGraphByReps}(G)$

Algorithm 34: OPERATOR V

The kernel condensation is computed in a series of steps, each implemented as a separate function. The main algorithm, above, uses these functions as follows.

First, M is initialized as an adjacency list with the same size as $G.A$, but with all neighbor lists empty. Then, GETBIPARTIDESINKREACHABILITYGRAPH, is called to create one edge from each non-sink to the sinks it reaches in the adjacency list M. If the resultant bipartite

graph has any edges, then at least one non-sink reaches a sink. The reduction continues by merging all non-sink vertices that reach exactly one sink with the sink that they reach (by calling MERGESINGLESINKREACHINGVERTICESWITHSINK).

Next, an order is obtained over each neighbor list of M, such that neighbor lists that are empty are placed first, and neighbor lists that are equivalent are placed adjacently in the order (using GETNEIGHBORLISTORDER). The index of the first non-empty neighbor list is obtained as i (with GETINDEXOFFIRSTNONEMPTYNEIGHBORLIST). Then, the order o is used together with the index of the first non-empty neighbor list i, to unify the disjoint sets of all vertices which have the same set of reachable sinks, using MERGEVERTICESWITHSAME-SINKREACHABILITY. Finally, SORTGRAPHBYREPS is called to unify the edge lists of all unified vertices in the disjoint sets.

We now describe the specifics of each of the functions called.

```
   Input: G
 1 Function GETBIPARTIDESINKREACHABILITYGRAPH(G):
 2 |   G.A = TRANSPOSE(G.A)
 3 |   for u ∈ G.D do
 4 |   |   if u ∈ G.S then
 5 |   |   |   M[u] = DFS(G, u)
 6 |   |   end
 7 |   end
 8 |   G.A = TRANSPOSE(G.A)
 9 |   M = TRANSPOSE(M)
10 |   return M
11 Function HASANYEDGES(M):
12 |   for 0 ≤ i < |M| do
13 |   |   if |M[i]| ≠ 0 then
14 |   |   |   return true;
15 |   |   end
16 |   end
17 Function MERGESINGLESINKREACHINGVERTICESWITHSINK(G, M):
18 |   for 0 ≤ i < |M| do
19 |   |   if |M[i]| = 1 then
20 |   |   |   G.D.UNIFYSETS(i, M[i][0])
21 |   |   end
22 |   end
```

Algorithm 35: GETBIPARTIDESINKREACHABILITYGRAPH, HASANYEDGES, and MERGESINGLESINKREACHINGVERTICESWITHSINK

GETBIPARTIDESINKREACHABILITYGRAPH transposes $G.A$, performs a DFS from each sink u to find all non-sink vertices v reached by u, and adds an edge from u to each v in M. Then M is transposed to direct edges from non-sinks to sinks, and returned by the function.

HASANYEDGES checks for the existence of any edge in M, by checking if any neighbor list of M is non-empty.

MERGESINGLESINKREACHINGVERTICESWITHSINK iterates over each neighbor list in M, then for any neighbor lists that contain only one vertex – meaning that the non-sink of that neighbor list reaches only one sink – the non-sink vertex is merged with the sink in the call to UNIFYSETS.

```
Input: G
Function GetNeighborListOrder(G, M):
    Function CompareNeighborLists(i, j):
        if |M[i]| ≠ |M[j]| then
            return |M[i]| < |M[j]|
        else
            for 0 ≤ k < |M[i]| do
                if M[i][k] ≠ M[j][k] then
                    return M[i][k] < M[j][k]
                end
            end
            return false
        end
    o ← list of size |G.A|s.t.∀0 ≤ i < |M| : o[i] = i
    for 0 ≤ i < |M| do
        M[i].sort()
    end
    sort_by(o, CompareNeighborLists)
    return o
Function GetIndexOfFirstNonEmptyNeighborList(G, M, o):
    i ← 0
    while |M[o[i]]| = 0 do
        i ← i + 1
    end
    return i
Function MergeVerticesWithSameSinkReachability(G, M, o, i):
    prev ← I[i]
    i ← i + 1
    while i < |o| do
        next ← o[i]
        if |M[prev]| = |M[next]| then
            j ← 0
            while j < |M[prev]| and M[prev][j] = M[next][j] do
                j ← j + 1
            end
            if j = |M[prev]| then
                G.D.UnifySets(prev, next) ;
            end
        end
    end
```

Algorithm 36: GETNEIGHBORLISTORDER, COMPARENEIGHBORLISTS, GETINDEXOFFIRSTNONEMPTYNEIGHBORLIST, and MERGEVERTICESWITHSAMESINKREACHABILITY

GETNEIGHBORLISTORDER first defines an internal comparator function COMPARENEIGHBORLISTS, which takes a pair of vertex identifiers i, j. The COMPARENEIGHBORLISTS function returns true if the neighbor list of $M[i]$ is to be ordered before $M[j]$, and false otherwise. This ordering is defined such that, if $M[i]$ is lesser in size than $M[j]$, then $M[i]$ occurs before $M[j]$; and if $M[i]$ is equal in size to $M[j]$, then the ordering of $M[i]$ and $M[j]$ is based on the comparison of the first non-equal neighbor at the same index in each, i.e. if $M[i][k] < M[j][k]$. Outside of the comparator, an ordering o is initialized to be the same size as $G.A$, with each member of the ordering assigned a value equal to its own index in that ordering. The neighbor lists of M are then sorted, and subsequently the comparator COMPARENEIGHBORLISTS function used to sort the neighbor lists themselves according to the ordering that function defines.

GETINDEXOFFIRSTNONEMPTYNEIGHBORLIST finds the index of the first non-empty neighbor list in M, according to an iteration order defined by o.

MERGEVERTICESWITHSAMESINKREACHABILITY iterates over each neighbor list in M, according to the ordering o. Each neighbor list $M[prev]$ is compared with those neighbor lists $M[next]$ that are immediately after it in the order o, and for any $M[next]$ equal to $M[prev]$, the disjoint sets of $prev$ and $next$ are merged.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	T	T	F	T	T	T

The post-conditions here are, again, the same as the pre-conditions.

A3.6.3 OPERATOR A

The final operator, **A**, randomly assigns a set vertices as sinks.

$$\neg G.\text{ifSinks} \rightarrow \text{TURNLEAVESINTOSINKS}(G)$$
$$\neg G.\text{ifAcyclic} \rightarrow \text{OPERATOR S}(G)$$
$$\neg G.\text{ifSinkReachable} \rightarrow \text{OPERATOR T}(G)$$
$$\neg G.\text{ifRootV} \rightarrow \text{ADDROOTVERTEX}(G)$$
$$\neg G.\text{ifBackwards} \rightarrow \text{REVERSEEDGES}(G)$$
$$\neg G.\text{ifRootE} \rightarrow \text{ADDEDGESFROMSINKSTOROOT}(G)$$

The pre-processing first ensures that the graph has sinks, by calling TURNLEAVESINTOSINKS to turn all vertices which have no out-neighbors into sinks. The pre-processing then ensures the graph is acyclic, that all non-sinks reach sink vertices, that the edges are backwards directed, and a root vertex exists with edges from the root to all sinks.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
T	T	T	T	T	T	T

The pre-conditions assert the requirements ensured by the pre-processing, and all graph state variables are here required to be true.

The algorithm randomly assign a selection of vertices in the graph as sinks, with the selection determined by arguments passed to `-extra`. **A** is the only operator to use the `-extra` CLI option. The option sets the population size from which to draw the random sample, the size of the random sample to be drawn, and whether to only consider existing sinks as part of the population, or all vertices up to the size of the population.

Input: G, so, ps, pr, ss, sr
1 $o \leftarrow$ TOPOLOGICALORDER$(G, 0)$
2 REMOVEEDGESFROMROOTTOSINKS(G)
3 $total \leftarrow$ GETTOTAL(G, so, o)
4 COMPUTEPOPULATIONSIZEANDRATIO$(ps, pr, total)$
5 COMPUTESAMPLESIZEANDRATIO(ss, sr, ps)
6 $population \leftarrow$ GETPOPULATION(o, ps)
7 TURNALLSINKSINTONONSINKS(G)
8 $sample \leftarrow$ GETRANDOMSAMPLE$(population, ss)$
9 TURNSAMPLEINTOSINKS$(G, sample)$
10 SORTGRAPHBYLABELS(G)
11 REVERSEEDGES(G)
12 CLEARALLSINKEDGES(G)

Algorithm 37: OPERATOR A

This algorithm is split into a number of stages. We first look at the inputs to the algorithm, as these are different from prior operators, and capture the arguments passed via the `-extra` CLI option.

The so parameter is a boolean flag, set in the CLI with `sinksOnly=1` to make so true and `sinksOnly=0` to make so false. If so is true, then the algorithm considers only existing sinks as part of the population from which to draw the sample of random sinks.

The ps parameter, set with `population=N` for some integer value of N in the CLI, determines the population size. If the population size is greater than the number of sinks, and the so parameter is true (so only sinks should be used as part of the population), then the population size used will be the number of sinks.

The pr parameter can be used instead of ps to determine the population size, by passing `population=N` with N as a floating-point value between 0.0 and 100.0. This population ratio pr is then used as the percentage of the total number of vertices which will be used as the population size if so is false, or, if so is true, the percentage of the number of existing sink-vertices used as the population size.

Similarly, the ss parameter, set with `sample=N` for an integer N, gives the size of the random sample to be drawn from the population. The sr parameter, set with `sample=N` for a floating-point N between 0.0 and 100.0, gives the ratio of the population to be used for the random sample.

We return to discussion of the algorithm implementing the operator **A** itself. Initially, a topological order is computed over G from the root vertex (with unique identifier 0), and this order assigned to o. The root edges are then removed by a call to REMOVEEDGES-FROMROOTTOSINKS. The value of $total$ is then obtained as the number of vertices from which to draw the population. If so is true, then the total returned by GETTOTAL is bounded by the number of sinks. The population size ps and population ratio pr are then computed using the $total$ with COMPUTEPOPULATIONSIZEANDRATIO. If the population parameter has been passed as a percentage (e.g. `population=50.0`) then pr is used to compute the population size ps (as 50% of $total$). Alternatively, if the population has been passed as an integer (e.g. `population=20`) then ps is used to compute pr (as $pr \times total$). The call to COMPUTESAMPLESIZEANDRATIO does similarly for the sample size ss and sample ratio sr, this time using the population size ps instead of the total.

The next call is to GETPOPULATION. The value of $population$ is obtained from the topological order o and the population size ps, such that vertices are selected for the population in the order that they occur in the top-sort. As the graph is backwards directed, the order o first contains all existing sinks and leaf vertices, followed by their set of inbound neighbors, subsequently followed by the set of inbound neighbors of these neighbors, and so on, and so forth. TURNALLSINKSINTONONSINKS changes the labels of the graph, such that all sinks become non-sinks. GETRANDOMSAMPLE uses the C++ standard library to select a random sample from $population$, using a 32-bit pseudo-random Mersenne Twister generator, with 19937 bits state size. The TURNSAMPLEINTOSINKS sets each label of the unique identifiers in $sample$ to be sinks. Finally, CLEARALLSINKEDGES removed outbound edges from all new sink vertices.

We now provide more detail on each of the subroutines involved in the random assignment of sinks is given hence.

```
1  Function GETTOTAL(G, so, o):
2      total ← 0
3      for 1 ≤ i < |o| do
4          if so and o[i] ∉ G.S then
5              break
6          else if o[i] ≠ ⊥ then
7              total ← total + 1
8          end
9          return total
10     end
11 Function COMPUTEPOPULATIONSIZEANDRATIO(ps, pr, total):
12     if ps = ⊥ and pr ≠ ⊥ then
13         ps = round(pr × total)
14     else if ps ≠ ⊥ and pr = ⊥ then
15         pr = ps/total
16     else
17         error
18     end
19 Function COMPUTESAMPLESIZEANDRATIO(ss, sr, ps):
20     if ss = ⊥ and sr ≠ ⊥ then
21         ss = round(sr × ps)
22     else if ss ≠ ⊥ and sr = ⊥ then
23         sr = (ss/ps)
24     else
25         error
26     end
```

Algorithm 38: GETTOTAL, COMPUTEPOPULATIONSIZEANDRATIO, and COMPUTESAMPLESIZEANDRATIO

The GETTOTAL computes $total$ by iterating over the vertices of G, according to the topological order o. If so is true, then the count for $total$ is stopped after the first non-sink encountered (as all remaining vertices in o are non-sinks by the ordering property). Otherwise, the $total$ count is incremented for each disjoint set representative encountered in the iteration over o. COMPUTEPOPULATIONSIZEANDRATIO uses ps (the population size) to define pr (the population ratio) if it is not yet defined, and pr to define ps if ps is not yet defined.

Similarly, COMPUTESAMPLESIZEANDRATIO computes the sample size ss via the sample ratio sr if ss is undefined, and sr by ss if sr is not defined.

```
Function GETPOPULATION(o, ps):
    ∀0 ≤ i < ps : population[i] = ⊥
    for 1 ≤ i < |o| do
        if o[i] − 1 < ps then
            population[o[i]] ← i
        end
    end
    return population
Function TURNALLSINKSINTONONSINKS(G):
    for 0 < i ≤ |G.L| do
        if G.L[i] ∈ G.S then
            G.L[i] ← "v_" + G.L[i]
        end
    end
Function TURNSAMPLEINTOSINKS(G, sample):
    for 0 < i ≤ |sample| do
        G.L[sample[i]] ← "h_" + G.L[i]
    end
Function CLEARALLSINKEDGES(G):
    for 0 ≤ i < |G.A| do
        if G.L[i] ∈ G.S then
            G.A[i].clear()
        end
    end
```

Algorithm 39: GETPOPULATION, TURNALLSINKSINTONONSINKS, TURNSAMPLEINTOSINKS, and CLEARALLSINKEDGES

GETPOPULATION iterators over o (from index 1 so as to disclude the root), and adds each vertex in order of the top-sort on the reverse graph to the population. TURNALLSINKSINTONONSINKS simply appends "v_" to the label of each sink, so as to make them a non-sink. TURNSAMPLEINTOSINKS goes through each unique identifier in the passed random sample, and appends "h_" to their label to transform them into sink vertices. Finally, CLEARALLSINKEDGES iterates through each sink vertex and clears the edge set thereof in $G.A$.

ifRootE	ifRootV	ifAcyclic	ifBackwards	ifInitialized	ifSinkReachable	ifSinks
F	T	T	**F**	T	**F**	T

The post-conditions are modified from the pre-conditions. The root edges have been removed (ifRootE is false), the graph is now forwards directed (ifBackwards is false), and it is no longer the case that all non-sinks necessarily reach sinks (so ifSinkReachable is false).