UNIVERSITY OF CALIFORNIA SAN DIEGO

A Declarative Framework for Big Graph Analytics and their Provenance

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Vasiliki Papavasileiou

2018

DEDICATION

To my family, Riccardo and Εκτοράκι.

EPIGRAPH

As you set out for Ithaka
hope your road is a long one,
full of adventure, full of discovery.
Laistrygonians, Cyclops,
angry Poseidon—don't be afraid of them:
you'll never find things like that on your way
as long as you keep your thoughts raised high,
as long as a rare excitement
stirs your spirit and your body.
Laistrygonians, Cyclops,
wild Poseidon—you won't encounter them
unless you bring them along inside your soul,
unless your soul sets them up in front of you.

Hope your road is a long one.
May there be many summer mornings when,
with what pleasure, what joy,
you enter harbors you're seeing for the first time;
may you stop at Phoenician trading stations to buy fine things,
mother of pearl and coral, amber and ebony,
sensual perfume of every kind—
as many sensual perfumes as you can;
and may you visit many Egyptian cities
to learn and go on learning from their scholars.

Keep Ithaka always in your mind.
Arriving there is what you're destined for.
But don't hurry the journey at all.
Better if it lasts for years,
so you're old by the time you reach the island,
wealthy with all you've gained on the way,
not expecting Ithaka to make you rich.

Ithaka gave you the marvelous journey.
Without her you wouldn't have set out.
She has nothing left to give you now.

And if you find her poor, Ithaka won't have fooled you.
Wise as you will have become, so full of experience,
you'll have understood by then what these Ithakas mean.

Ithaka by Constantine Peter Cavafy

TABLE OF CONTENTS

Signature Page iii

Dedication iv

Epigraph v

Table of Contents vi

List of Figures viii

List of Tables xi

Acknowledgements xii

Vita xiv

Abstract of the Dissertation xv

Chapter 1 Introduction 1
- 1.1 Overview 3
  - 1.1.1 Datalography: Declarative Big Graph Analytics 3
  - 1.1.2 Ariadne: Online Provenance Querying for Big Graph Analytics . . . 4
  - 1.1.3 Contributions 5

Chapter 2 Datalog 7
- 2.1 Introduction 7
- 2.2 Program evaluation 8
- 2.3 Datalog recursion with aggregates 10
  - 2.3.1 Stratified semantics 10
  - 2.3.2 Monotonic aggregation 14

Chapter 3 Vertex-Centric Graph Processing 17
- 3.1 Introduction 17
- 3.2 Vertex-Centric 17
- 3.3 Think like a graph 19

Chapter 4 Declarative Graph Analytics on Vertex Centric Engines 21
- 4.1 Introduction 21
- 4.2 Declarative Big Graph analytics 23
- 4.3 Efficient recursive aggregation 24
- 4.4 VC-Datalog 26
  - 4.4.1 Vertex-Centric normal form 26
  - 4.4.2 Vertex-Centric rewriting 28
  - 4.4.3 Compilation and Planning 36

4.5 Optimizations . . . 39
4.5.1 Compilation . . . 41
4.5.2 Super-vertices . . . 42
4.5.3 Combiners . . . 44
4.6 Implementation . . . 47
4.7 Experimental evaluation . . . 48
4.7.1 Experimental setup . . . 48
4.7.2 Runtime analysis . . . 51
4.7.3 Scaling super-vertices . . . 55
4.8 Related work . . . 59
4.9 Chapter summary . . . 65

Chapter 5 Ariadne: Online Provenance Querying For Big Graph Analytics . . . 69
5.1 Introduction . . . 69
5.2 Running example and System overview . . . 71
5.3 Provenance model . . . 75
5.4 Provenance querying . . . 78
5.4.1 Provenance Query Language . . . 78
5.4.2 Layer-at-a-time evaluation . . . 79
5.5 Experimental evaluation . . . 85
5.5.1 Capturing overheads . . . 86
5.5.2 Online provenance querying . . . 88
5.5.3 Offline provenance querying . . . 97
5.6 Related work . . . 100
5.7 Chapter summary . . . 101

Chapter 6 Conclusion and Future Directions . . . 103

Bibliography . . . 105

LIST OF FIGURES

Figure 2.1. The dependency graph of Query 1. 10

Figure 4.1. Running example: Imperative implementation of PageRank on Giraph 24

Figure 4.2. Steps followed to transform a general Datalog query into a *Vertex-Centric* query 31

Figure 4.3. The graph representation of $body(R)$. 31

Figure 4.4. The Routing-Tree of R denoting how vertices will communicate during query evaluation. 32

Figure 4.5. Edge-Tree annotated with predicates annotating its vertices and edges 34

Figure 4.6. The dependency graph for running example Query 11. 37

Figure 4.7. Desired dependency graph of Query 5 that captures distribution. . . . 38

Figure 4.8. The dependency graph of Query 11. 40

Figure 4.9. The logical plan for Query 11. 40

Figure 4.10. Example: Partitioning of input graph into super-vertices 42

Figure 4.11. Schema of super-vertex SV_2 43

Figure 4.12. Messages sent with eager-aggregation 46

Figure 4.13. DATALOGRAPHY architecture 47

Figure 4.14. SSSP: Performance comparison of Giraph, Giraph-Metis and DATALOGRAPHY 52

Figure 4.15. WCC: Performance comparison of Giraph, Giraph-Metis and DATALOGRAPHY 54

Figure 4.16. PageRank: Performance comparison of Giraph, Giraph-Metis and DATALOGRAPHY 55

Figure 4.17. SSSP IN-04: Runtime and communication per super-vertex size. 57

Figure 4.18. SSSP UK-02: Runtime and communication per super-vertex size. . . . 58

Figure 4.19. SSSP AR-05: Runtime and communication per super-vertex size. . . . 59

Figure 4.20. SSSP UK-05: Runtime and communication per super-vertex size. . . . 60

Figure 4.21. WCC IN-04: Runtime and communication per super-vertex size. 61

Figure 4.22. WCC UK-02: Runtime and communication per super-vertex size. . . . 62

Figure 4.23. WCC AR-05: Runtime and communication per super-vertex size. . . . 63

Figure 4.24. WCC UK-05: Runtime and communication per super-vertex size. . . . 64

Figure 4.25. PageRank IN-04: scaling super-vertices 65

Figure 4.26. PageRank UK-02: scaling super-vertices 66

Figure 4.27. PageRank AR-05: scaling super-vertices 67

Figure 4.28. PageRank UK-05: scaling super-vertices 68

Figure 5.1. Running example: Giraph SSSP 72

Figure 5.2. Architecture of Ariadne that improves traditional offline provenance usage. 73

Figure 5.3. Online provenance querying using Ariadne. 74

Figure 5.4. Running example: Giraph SSSP with convergence criterion 75

Figure 5.5. Provenance graph for SSSP 76

Figure 5.6. Compact provenance graph 77

Figure 5.7. Provenance graph of SSSP divided into three layers. 81

Figure 5.8. Information flow in evaluation of R_1 81

Figure 5.9. Information flow in directed queries 82

Figure 5.10. Vertex x at superstep i evaluates vertex program and PQL query. . . 83

Figure 5.11. Runtime overhead of Query 17 and Query 18. 89

Figure 5.12. Running time of PageRank audit queries. 91

Figure 5.13. Running time of SSSP and WCC audit queries. 92

Figure 5.14. Running time for ALS audit queries. 94

Figure 5.15. Runtime improvement between original (Giraph) and optimized (Giraph - opt) analytic 96

Figure 5.16. Running time of running example Query 16 97

Figure 5.17. Runtime overhead for backward lineage query 26 99

LIST OF TABLES

Table 4.1. Notations used 28

Table 4.2. Dataset characteristics 49

Table 4.3. Number of super-vertices created for every dataset. 56

Table 5.1. Provenance EDB predicates 79

Table 5.2. Dataset characteristics 86

Table 5.3. Size comparison of input graph and full provenance graph 87

Table 5.4. Size comparison of input graph and tailored provenance graph 88

Table 5.5. PageRank: Relative error (L_2) for $\epsilon = 0.01$ and median values of original (A) and optimized (B) analytics. 95

Table 5.6. SSSP: Relative error (L_1) for $\epsilon = 0.1$ and median values of original (A) and optimized (B) analytics. 96

ACKNOWLEDGEMENTS

I want to thank Alin Deutsch for his guidance and support. He has been an inspirational example of personal and professional integrity. Always setting the bar high, he urged me to improve my ideas and demonstrated confidence in my abilities to solve tough problems. Additionally, I would like to thank Ken Yocum for without him this dissertation would have taken twice as long. Despite having a full time position in industry, he kept active involvement in my research and his commitment to this work kept me motivated through the tough times.

I would also like to thank Walaa Eldin Moustafa, my mentor during my internship at NEC Laboratories. Our collaboration gave the seeds that grew into this dissertation.

I am grateful to my committee Tyson Condie, Gert Lankriet for their insightful comments and George Porter who graciously allowed me to use machines from the Sysnet cluster for my experiments.

I would like to thank the members of the DB lab at UCSD, Mohan Yang for answering my questions about his work while at UCLA, Cindy Moore for administering the cluster and accommodating my requests for extensions and Julie Conner for all her help and support.

This journey would not be nearly as enjoyable without the great people I met in San Diego and am lucky to call my friends. They made San Diego feel like home and became my family. I am especially grateful for Nevena, Yiannis, Dimo and Andreja who supported and stood by me when I was going through really difficult times. You will forever be part of my life. Also, Vineet (my roommate and initiator in spicy food), Karyn (her positive attitude and encouragement that I was not alone in my struggles), Akshay (his intelligence and kindness made us friends since the first day at UCSD), Valentin, John, Niki, Louis, and the rest of the wonderful grad students that make the CSE community unforgettable. I am grateful for the small but strong community of Greeks at UCSD (and outside) and especially Andreas, Kostas, Nikos, Panos, Ioanna, Stamatia, Stavros, Petros, Andreas that were part of my every-day life. Moreover, I thank Mikael and Jaymee and the rest of the friends in Awesome San Diego for the awesome parties and Wednesday dinner gatherings.

I am grateful to my family who endured hardships and sacrifices in their life so that I could have a better one. Special thanks to my grandparents Athanasios and Vasiliki who raised me and showered me with love, my mother Sofia for her unconditional love and my sister Georgia, my best friend in life.

Finally, I owe this dissertation to Riccardo Beltramo. His faith and pride in me inspire me to be a better person. His love and support has been a driving force behind this PhD and I could not have done it without him.

Chapter 4, in part, contains material from ”Datalography: Scaling datalog graph analytics on graph processing systems” by Walaa Eldin Moustafa, Vicky Papavasileiou, Ken Yocum and Alin Deutsch, which appears in Proceedings of International Conference on Big Data 2016. The dissertation author was a primary investigator and co-author of this paper.

Chapter 5, in full, has been submitted for publication of the material. Vicky Papavasileiou, Ken Yocum, Alin Deutsch. The dissertation author was the primary investigator and author of this paper.

VITA

2002-2007	Bachelor of Science, Computer Science Department, University of Crete Greece
2007–2009	Master of Science, Computer Science, University of Crete Greece
2010–2018	Research Assistant, University of California San Diego
2018	Doctor of Philosophy, University of California San Diego

ABSTRACT OF THE DISSERTATION

A Declarative Framework for Big Graph Analytics and their Provenance

by

Vasiliki Papavasileiou

Doctor of Philosophy in Computer Science

University of California San Diego, 2018

Alin Deutsch, Chair
Ken Yocum, Co-Chair

Recent years have witnessed an explosion in size of graph data and complexity of graph analytics in fields such as social and mobile networks, science and advertisement. Analyzing and extracting knowledge from Big Graphs (in analogy to Big Data) is hard. The size of Big Graphs necessitates the use of distributed infrastructures and parallel programming. Moreover, implementing performant and correct analytics requires in depth knowledge of both algorithm and input data. Developers of graph analytics face two major challenges: i) There is a myriad of Big Graph processing frameworks, each uses a different imperative programming language and implements different low-level optimizations. Developers are burdened with understanding the low-level characteristics of an execution framework that suits best their algorithms and data. ii) Assessing the quality of both data and analytics is a tedious and manual task. Devising new graph analytics is an iterative process, where developers incrementally refine their algorithms and clean their data by analyzing results, correcting for errors and run again until the end results are satisfiable.

In this dissertation we offer a declarative framework that addresses the entire life-cycle, from designing to executing, of Big Graph analytics. Our approach uses a single language for both authoring graph analytics and fine-tuning them. Specifically, this dissertation makes the following two main contributions:

We design and demonstrate Datalography, the first approach for declarative graph analytics on *Vertex-Centric* graph processing engines. To accommodate different programming models, we design and implement a compiler that takes general Datalog queries and rewrites them into distribution-aware queries that can be efficiently evaluated on any *Vertex-Centric* framework. Moreover, our compiler implements automatic and transparent to the user optimizations in the form of logical query rewritings and thus are portable to any *Vertex-Centric* system. We demonstrate the effectiveness of our approach with an experimental evaluation on real-world graphs that indicates Datalography offers superior performance when compared to native, imperative implementations.

Our second contribution is a novel provenance management approach that enables developers to customize provenance capturing and analysis with twofold benefits: the amount of captured provenance is minimized to include only the necessary information and analysis is extended beyond the traditional tracing queries. We present formal semantics of our provenance query language, based on Datalog, and identify an important class of queries that can be evaluated *online*, simultaneously with the graph analytic. We showcase our approach with Ariadne, a provenance management system that supports efficient debugging, auditing and fine-tuning of graph analytics.

Chapter 1

Introduction

Recent years have seen an explosion in size and variability of graph data in fields such as social and mobile networks, advertisement and science. Graphs represent succinctly natural phenomena where vertices denote entities and edges denote relationships between them. We use graphs in our every day life. For example, when we connect with friends using social networks, when we shop online and receive targeted recommendations and ads or when we use mobile GPS devices that suggest the shortest route. In science, graphs are used for example to uncover gene-drug interactions, in DNA assembly, and in mapping the human brain connectome. The sizes of these graphs are constantly increasing: The number of active Facebook users has doubled from 1.3 billion in 2014 to 2.3 billion in 2018. The web pages in Google's index are 40 billion in 2018. The human brain connectome consists of at least 100 billion vertices and 100 trillion edges.

Analyzing and extracting knowledge from Big Graphs (in analogy to Big Data) involves performing graph queries such as reachability, shortest paths, and measuring connected components, as well as more sophisticated computations such as training machine learning models (recommenders, classifiers, etc.).

Big Graphs necessitate the use of distributed infrastructures. The growing interest of researchers with different backgrounds to analyze Big Graphs has lead to a flurry of Big Graph processing frameworks([48, 1, 46, 31, 30, 61, 42]) that assist users in developing and executing distributed and parallel analytics at scale. In their majority, they follow the *Vertex-Centric* programming model that offers a user-friendly API to shield developers from synchronization and distribution primitives. In theory, developers need not know the low-level implementation details of the framework they use. In practice, however this is not true especially when one is interested in the performance of their graph analytics. This is suggested by the myriad of VC frameworks in existence that each use a different imperative programming language and implement different low-level optimizations. Some even propose different programming APIs beyond the *Vertex-Centric* model [73, 70, 67, 69]. There is no one system that incorporates all optimizations and several surveys [80, 39, 72, 24, 34, 36, 47] aim to categorize the benefits and drawbacks of each framework.

Developers must have broad knowledge of the systems out there and their low-level engineering quirks, to select the system that is best suited to their algorithms and data. On top of that, they must manually optimize the execution through coding practices to improve performance. This results in a lot of duplicate effort as system optimizations transcend systems and analytic's optimizations transcend analytics. There is the need for a high-level abstraction that provides a universal way to reason about analytics and their data independent of the underlying system and programming language. A declarative language fills this gap as it provides a common front-end to developers while a declarative compiler can choose and implement a host of optimizations, *automatically* and *transparently* to the user. Finally, a declarative language that naturally supports graph queries is amenable to *concise* specification of graph analytics.

After developing Big Graph analytics, common practice is to analyze their behavior for fine-tuning. This involves correcting for errors such as fixing bugs and data cleaning, improving the quality of results with model training and parameter regularizations, improving the performance by skipping execution when convergence is reached and finally, monitoring execution as new data arrives. Thus, implementing new graph analytics is an iterative process, where developers incrementally refine their algorithm and clean their data by analyzing results, correcting for errors and run again until the end results are satisfiable. Assessing the quality of both data and analytics is currently a tedious and manual task. Much like provenance management in RDBMS and scientific workflows, there is the need for provenance management on Big Graph processing systems that assist developers in their debugging, auditing and fine-tuning needs.

1.1 Overview

In this thesis, we provide a single language solution to authoring, optimizing and analyzing Big Graph analytics. A high-level, declarative language allows to succinctly express *what* graph analytics compute and not *how* to compute it. Much like what SQL achieved for database systems, a declarative graph language moves the burden of optimization from the user to the language compiler. Datalog is a natural fit as it supports recursion, aggregation and there is vast research on optimization techniques for query evaluation.

1.1.1 Datalography: Declarative Big Graph Analytics

Recent efforts have suggested using Datalog for authoring Big Graph analytics. [62, 71] built bespoke distributed datalog evaluation engines that however, do not fit in Big Data workflows. Graph analytics, more often than not, are part of a workflow that integrates different systems and varying data sources. Another approach, followed by [65] is to extend existing dataflow systems. They involve significant code changes to the underlying system to support efficient recursion and hence their solutions are not applicable to other systems. None of the aforementioned approaches addresses declarative graph analytics on graph processing systems that naturally support recursion.

Datalography supports the evaluation of *general* Datalog queries on *Vertex-Centric* engines. Developers need not worry about the programming model of underlying engines when expressing their graph analytics. In fact, all queries expressed by prior work are supported in DATALOGRAPHY without modifications. We provide a compiler that, behind the scenes, rewrites general Datalog queries to vertex-centric ones so that they can be efficiently evaluated on *Vertex-Centric* engines. Moreover, we provide novel logical optimizations, namely super-vertices and source-side combiners, that reduce the amount of computation and communication. We conduct an experimental evaluation that shows the benefit of our approach over imperative code and quantifies the gains achieved by each optimization.

1.1.2 Ariadne: Online Provenance Querying for Big Graph Analytics

Traditionally, provenance has been stored in databases (whether relational, XML or RDF) and queried offline using the query language provided by the respective data model of the database. Provenance capturing overheads were not the focus of research as provenance sizes were in tandem to input sizes and provenance was queried in an offline manner. Provenance, thus, was long-lived and the capturing overhead was amortized by the duration the stored provenance remained relevant. The advent of Big Data and large-scale data processing changes these assumptions as both input data and processing are transient, analytics execute multiple times with different parameters and on different data sets. Moreover, processing is distributed making provenance capturing a distributed problem. Finally, provenance sizes are multiple times larger than the input data and choosing database for storing provenance is not the defacto solution.

These challenges fueled new research for provenance management on Big Data analytics ([44, 55, 38, 7]), however all of them focus on batch-processing dataflow engines and provide limited querying capabilities in the form of offline, imperative traversal of the provenance graph.

ARIADNE is a system to capture, store, and analyze provenance for *Vertex-Centric* graph processing engines. ARIADNE defines a provenance querying language (PQL) that developers use to declaratively customize provenance capturing and analysis. Ariadne enables an important class of queries that allow *online* provenance analysis that executes simultaneously with an analytic. At the end of the computation, the results of the analytic and the provenance queries (capturing and analysis) exist. Moreover, because Ariadne represents the captured provenance as a graph, it is immediately available for further, repeat analysis on the same engine.

A high-level provenance query language along with online evaluation enables novel uses of provenance beyond the traditional debugging. We experiment with audit queries that monitor the computation and tuning queries that explore patterns in the computation to optimize the runtime of an analytic. We evaluate Ariadne on four graph analytics on real world graphs. When Ariadne evaluates the analytic simultaneously with multiple provenance queries (capturing and analysis) the overhead is $2x$ over the baseline Giraph analytic. When Ariadne evaluates only analysis queries without capturing additional provenance, the overhead is $1.3x$.

1.1.3 Contributions

In this dissertation we offer a declarative framework that addresses the entire life-cycle, from designing to executing, of Big Graph analytics. Our approach uses a single language, Datalog, for both authoring graph analytics and fine-tuning their behavior. In the first part of the dissertation we make the following contributions:

- We implement a declarative compiler and execution engine that follows the *Vertex-Centric* paradigm. Our solution does not require changes to the *Vertex-Centric* system and is portable to other systems following the same paradigm.

- We design a compiler that takes general Datalog queries and rewrites them into VC-Datalog queries that conform to the *Vertex-Centric* paradigm.

- We propose the logical abstraction of super-vertices that enable set-at-a-time evaluation, without changing the user API.

- We implement optimizations, applied automatically and transparently to the user, that reduce the amount of communication overhead.
- Our experiments show that Datalog graph analytics outperform imperative ones by a factor of $1.4x$-$7.8x$.

In the second part of the dissertation, we use our declarative framework to address provenance capturing and querying of Big Graph analytics for scenarios extending the traditional crash-culprit determination. We make the following contributions:

- We provide a formal provenance model, the Provenance Graph, that enables querying provenance using the same *Vertex-Centric* system the analytic ran.
- We define a Provenance Query Language, based on VC-Datalog, for querying the provenance graph.
- We propose the novel *layer-at-a-time* evaluation method that is scalable as it does not require materializing of the entire provenance graph.
- We propose the novel *online* evaluation mode that obviates the need for capturing provenance.
- Our experiments show that online provenance querying incurs $1.3x$ overhead.

Chapter 2

Datalog

2.1 Introduction

A Datalog program or query consists of a set of rules and a set of facts. Facts represent statements that are true, whereas rules allow us to deduce new facts from other known or previously deduced facts. A Datalog rule has the following syntax:

$$L_0 \leftarrow L_1, ..., L_n$$

where each L_i is a literal of the form $P_i(x_1, \ldots, x_n)$, where P_i is a predicate, and $x_1, \ldots, x_n$ are terms. Terms can be variables, constants or functions. Informally, rules are read as "If $L_1, \ldots, L_n$ are true, then L_0 is true." L_0 is the head of the rule, and $L_1 \ldots, L_n$ is the rule's body. Datalog rules can be recursive, i.e., a rule can recursively depend on itself or another rule that depends on it.

A fact is a rule with an empty body and is always true. A fact that has all its terms constant is called a ground fact. A Datalog program begins with an extensional database (EDB) containing ground facts. In database terminology, EDB is the input database, each predicate corresponds to a relation (table) and facts correspond to tuples. For example, a program can have an EDB that models a graph consisting of two relations, namely the vertices and edges. Evaluation of a Datalog program populates the input database with new relations and tuples, called the intentional database (IDB).

We illustrate Datalog with a simple program that finds all vertices reached, transitively, from a source vertex. Program 1 has three rules: Rule `start` uses the **vertex** predicate that denotes the vertices in a graph and identifies the vertex from which to start the reachability query. Rule `reach` has two bodies: the first uses the vertex in relation **start** and produces new **trace** facts for every outgoing neighbor of it, identified by predicate **edges**. The second body is recursive as predicate **trace** appears both in the head and in the body and produces new **trace** facts from previously computed ones and their outgoing neighbors.

$$\begin{aligned}&\mathbf{start}(x) \leftarrow \text{vertex}(x), x == 0.\\&\mathbf{reach}(x) \leftarrow \text{start}(x).\\&\mathbf{reach}(x) \leftarrow \text{edges}(y, x), \text{trace}(y).\end{aligned}$$

Query 1. Reachability

2.2 Program evaluation

Under the least fixpoint semantics of Datalog, a recursive program is evaluated using the *bottom-up* evaluation strategy. Intuitively, the rules in a program are repeatedly invoked on existing tuples until no more new tuples can be derived. A simple algorithm for this, coined *naive* evaluation proceeds in iterations: In the beginning, the program is invoked on the EDB, in our case the vertices and edges of the graph and the IDB is initialized to $\emptyset$. In our reachability example, the only new facts that can be derived in one iteration are $\mathbf{start(0)}$ and $\mathbf{reach(0)}$ which are added to the IDB. In every subsequent iteration, the rules are invoked on the EDB as well as the previous IDB. When there are no more changes to the IDB, i.e., no more new tuples added, evaluation stops.

Naive evaluation is inefficient as it may produce the same facts multiple times. For example, a vertex may be reachable from the source using multiple paths, and naive evaluation will deduce the same, previously discovered fact multiple times, once for each such path. However, only one of the tuples is part of the result as Datalog assumes set-semantics.

The *semi-naive* [10] evaluation improves upon this by exploiting the fact that during naive evaluation the EDB relations remain unchanged and new facts are deduced only from the new facts in the IDB of the previous iteration. Thus, instead of repeatedly evaluating the program using the entire set of tuples found so far in the IDB, evaluation proceeds by using only newly discovered facts, referred to as the delta. This process is shown in Algorithm 1. I is the input EDB database, S contains all the deduced facts in IDB and ΔS^i contains all the newly deduced facts at every iteration i. In the beginning, only the rules that have exclusively EDB predicates in their body (P_0) are evaluated on the input database and their facts are added to ΔS^0 (line 3). Then, at every subsequent iteration, the program is evaluated on the delta of the previous iteration and the new delta is obtained by subtracting from the new facts all the previously known facts (line 6). This new delta is added to the IDB database S (line 7). Evaluation proceeds until the delta at an iteration is empty.

Algorithm 1. Recursive Semi-naive Datalog Evaluation

```
1: $S = \emptyset$
2: $P_0$ rules in $P$ without IDB predicates in their body
3: $\Delta_S^0 = P_0(I)$
4: $i = 1$
5: do
6:     $\Delta_S^i = P(\Delta_S^{i-1}) - S$
7:     $S = S \cup \Delta_S^i$
8:     $i = i + 1$
9: while $\Delta_S^i = \emptyset$
```

Notice, how there has been no mention on how the rules of the program are evaluated. Datalog does not impose an ordering on the rules of a program and assumes they are evaluate *in parallel.* In practice, however, an order is chosen such that when a rule evaluates, all the predicates in its body have already evaluated. The *dependence*

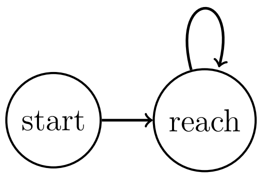


Figure 2.1. The dependency graph of Query 1.

graph is helpful to order rules in recursive programs: it contains vertices for every IDB predicate in the program and edges denote dependencies between these predicates. An edge is added between v_2 and v_1 if the predicate of v_2 appears in the body of a rule with head v_1. If the program is recursive, the dependency graph will contain cycles. Evaluation proceeds following a topological sorting of the graph, recursing on every strongly connected component. The dependency graph for our reachability query 1 is shown in Figure 2.1 where recursion is denoted with a self loop on predicate **reach**.

2.3 Datalog recursion with aggregates

The above simple semantics hold for monotone, positive Datalog programs. Problems arise when negation and aggregation are introduced as monotonicity with respect to set containment, and hence a minimal least fixpoint, is not anymore ensured. However, negation and aggregation are fundamental in expressing real-world applications and especially graph analytics – many compute until an aggregate value, such as a MIN, MAX, SUM, or COUNT, reaches a certain threshold.

2.3.1 Stratified semantics

Stratification allows negation in a Datalog program, as long as there is no recursion through it. The idea is to divide the rules of a program into *strata*, groups of rules, and for evaluation to proceed following an ordering on the strata whereby the IDB predicates of a previous stratum are considered as EDB and therefore unchanged. There is an easy syntactic check to verify if a program is stratifiable based on the dependency graph not having negative cycles. For this, the edges of the graph carry negative labels, whenever a negated IDB appears in the body of a rule. Moreover, to ensure termination, rules must be *safe* having every variable in the body appear in at least one positive predicate.

The same idea can be applied to programs with aggregation. [53] defines the "aggregate stratified" class of programs that, similar to the stratified negation class, disallows recursion through aggregation. Additionally, rules must be *range restricted* having every variable in the head appear in the body as well. Aggregation follows similar syntactic semantics as with SQL: the head has one or more grouping variables and a list of aggregate variables that appear in the aggregation function. The aggregate variables may not appear else in the head.

Stratified Datalog queries with aggregates, although correct, are inefficient. Consider for example Query 2 computing the shortest paths from a given source (vertex with id 0) without the use of aggregation in recursion. Besides this program not terminating in the presence of cycles in the input graph, it is also extremely inefficient as it first computes the length of all paths from the source and only then is the minimum for every vertex selected via the MIN aggregate outside the recursion (rule `sssp`).

$$\begin{array}{l}
\mathbf{path}(x,0) \leftarrow \text{vertex}(x), x == 0. \\
\mathbf{path}(x,d) \leftarrow \text{edges}(y,x,w), \text{path}(y,d_1), d = d1 + w. \\
\mathbf{sssp}(x, MIN(d)) \leftarrow \text{path}(x,d).
\end{array}$$

Query 2. SSSP: Recursion without aggregation

Several works suggested extensions to stratified semantics such as locally stratified programs [56], well-founded models [26] and stable models [28]. Here, we focus on one such approach, the *modularly stratified* [57] class that allows negation in recursion under constraints while still preserving two-valued semantics. The idea is to divide a program into strongly connected components and recursion is allowed only within a component. Evaluation proceeds component-at-a-time and only when the current component has converged, will the next one start executing. The problem is that checking whether a program is modularly stratified (and hence has a perfect model) is not anymore a syntactic check but depends on how rules get instantiated at *runtime* for a given input database. Although this class is the most expressive while still providing perfect model semantics, it requires sophisticated compilers and there are no efficient evaluation techniques.

This gives rise to two desiderata when choosing semantics to support negation and aggregation in recursion:

- There must exist simple syntactic checks to decide whether a program is allowed or not.
- Evaluation must be efficient and not compute unnecessary facts that will later be discarded.

These problems are addressed by [41] and [77] that each define a subclass of modularly stratified programs that allow negation and aggregation in recursion and whose stratifiability is verified by an easy syntactic check. Programs in this class are *explicitly* stratified with the use of a temporal variable in the IDBs: Non-recursive predicates are annotated with the constant 0 whereas all other predicates are annotated with i or $s(i)$, the succinct of i. The temporal variable essentially assigns strata level to IDB predicates.

Following [77], the recursive rules in a program are categorized into two groups: the *X-rules* where the temporal variable in their head is the same as of all predicates in their body and the *Y-rules* whose head has the temporal variable $s(i)$ and at least one positive predicate in the body has the temporal variable i. There is a simple syntactic to verify whether a program is xy-stratified and involves relabeling all head predicates by prepending them with the label "new" and labeling all predicates in the body with "new" as well if they have the same temporal variable else with "old". The initial program is xy-stratified if the re-labeled program is stratified, i.e. the dependency graph has no negative or aggregated cycles. The fixpoint evaluation of xy-stratified programs proceeds in iterations and involves replacing the tuples of the "old" relation with the "new" ones at every iteration.

Although XY-stratifiable semantics support negation and aggregation and allow for an easy-syntactic check, they can lead to cumbersome programs to write and inefficient execution. Consider again SSSP, this time expressed as an XY-program in Query 3. The query simulates semi-naive evaluation in a quite procedural way as the user has to specify how the delta is computed at every iteration (in multiple steps) and how to update the total set of facts using the delta (again multiple steps).

$r_1 : \textbf{cand-sp}(x, MIN(D), i+1) \leftarrow \text{delta-sp}(y, d_1, i), \text{edges}(y, x, d_2), d = d_1 + d_2.$
$r_2 : \textbf{delta-sp}(x, 0, 0) \leftarrow \text{vertex}(x), x == 0.$
$r_3 : \textbf{delta-sp}(x, d, i+1) \leftarrow \text{cand-sp}(x, d, i), \text{all-sp}(x, d_2, i), d_2 > d$
$r_4 : \textbf{delta-sp}(x, d, i+1) \leftarrow \text{cand-sp}(x, d, i), \neg\,\text{all-sp}(x, , i)$
$r_5 : \textbf{all-sp}(x, d, i+1) \leftarrow \text{all-sp}(x, d, i), \neg\,\text{delta-sp}(x, , i+1)$
$r_6 : \textbf{all-sp}(x, d, i) \leftarrow \text{delta-sp}(x, d, i)$
$r_7 : \textbf{last-sp}(x, i) \leftarrow \neg\,\text{delta-sp}(x, , i)$
$r_8 : \textbf{sssp}(x, d) \leftarrow \text{all-sp}(x, d, i).$

Query 3. SSSP: XY-stratified recursion with aggregation

Predicate **delta-sp** contains the shortest paths of vertices visited at the current iteration whereas predicate **all-sp** contains the shortest paths of all the vertices visited in the iterations up to the current one. The shortest path of a vertex at an iteration is computed in two steps: first, rule `cand-sp` computes a candidate shortest path based on the minimum distance of its incoming neighbors , then a candidate path becomes the shortest distance if there is no other path for that vertex (rule r_4) or if there is, it is longer (rule r_3). Finally, the shortest paths of all visited vertices (**all-sp**) are updated at every iteration to contain the distance for all vertices not visited in the current iteration (rule r_5) and the new distances for all vertices visited at the current iteration (rule r_6). Rule r_7 ensures that evaluation stops for a vertex when the delta is empty and rule r_8 chooses the shortest distance as the lastly computed one.

2.3.2 Monotonic aggregation

The previous approaches based on stratification provide uniform semantics for negation and aggregates, as well as other constructs such as arithmetic functions. They extend Datalog semantics to allow for the non-monotonicity of aggregates and negation but introduce complexities and prevent using optimizations such as semi-naive evaluation and magic sets.

Aggregation in recursion is unavoidable when one is interested in succinct formulation and performant evaluation. Monotonic aggregation achieves this for aggregate functions that are *monotonic*: If we look at the values produced by an aggregate, they are monotonically decreasing/increasing. This is true for instance for the MIN aggregate in SSSP which allows to express it as Query 13. The desire for monotonic aggregates inspired a flurry of research [25, 68, 19, 26] however, the correctness of semantics and evaluation are still an open debate [78, 79] and research has not yet converged to a widely accepted solution. Below we provide an overview of the two main lines of work for incorporating aggregation in recursion: lattices based on partial orders and continuous aggregates.

$$\begin{array}{l} \mathbf{sssp}(x,0) \leftarrow \text{vertex}(x), x == 0. \\ \mathbf{sssp}(x, MIN(d)) \leftarrow \text{edges}(y,x,w), \text{sssp}(y,d_1,i), d = d1 + w. \end{array}$$

Query 4. SSSP: Monotonic aggregation in recursion

First [58] suggested defining a partial order over the domain of an aggregate function that together with a meet and join operation, form a complete lattice. Monotonicity is then ensured when the ordering on the domain of the aggregate function transfers to its range. However, [27] showed that it is very difficult to identify automatically which lattices make a program monotonic and it is not trivial to formally define bottom-up evaluation. Recently [20], the extension of Bloom [6] language for declarative distributed programing, brought monotonicity based on partial orders to the foreground again. It improves upon prior work by providing a framework for developers to *safely* define lattices and proved that optimizations such as semi-naive evaluation and magic sets are applicable. However, the burden to prove the correctness of the chosen lattice falls again on the user.

The idea of lattices is desirable due to its simple idea and generality, users can define their own lattices and aggregate functions, as well as due to its efficiency, [62] showed Datalog programs outperform imperative hand-written code. However, the difficulty of providing formal proofs and guarantees has not let to a wide adoption of this approach.

Yet a different idea to handle aggregation are *continuous* aggregates [51, 50, 66] that avoid the above shortcomings by defining monotonicity with respect to set-containment. When using lattices, the number of tuples associated with a grouping key is at most one and when a new tuple is computed it replaces the previous one according to the definition of the *merge* operation of the aggregation function. On the contrary, with continuous aggregates, a growing set of tuples is associated with a grouping key. For query SSSP for example, assume vertex a with distances 5 at iteration 1, distance 3 at iteration 2 and distance 2 at iteration 3. Then relation **sssp** for grouping key a will have the following tuples after 3 iterations: $(a,5),(a,3),(a,2)$. We see that when a new tuple is computed for a grouping key, it is appended to its set of tuples and this is how set-containment is maintained. Monotonicity is defined then on the values an aggregate function produces for a specific grouping key: if the values when viewed as a sequence across all iterations are monotonically increasing/decreasing, then the aggregate function is monotonic. Obviously, the end result of aggregation must be a single tuple per grouping key and this approach requires that, at the end of evaluation, a stratified aggregation is applied to select the actual final value.

At first hand, this evaluation strategy is inefficient and doesn't differ from stratified evaluation in the amount of intermediate tuples produced. Luckily, the monotonicity of aggregate functions allows for optimizations. For MIN/MAX aggregates, we know their values are monotonically decreasing/increasing. Moreover, at every iteration during evaluation, the intermediate tuples other than the minimum/maximum can be safely ignored as they won't influence the final result. This gives rise to an evaluation strategy that does not need to maintain a set of intermediate tuples but only the latest one. Essentially, the optimized evaluation simulates the evaluation with lattices where also only one intermediate tuple is maintained. The COUNT/SUM aggregates are handled similarly: At every intermediate iteration only the maximum count/sum for a grouping key is maintained and the current aggregate value is updated using the difference between the previous and current maximum values.

Based on the above analysis, in this dissertation we follow the semantics of [77] for non-monotonic programs whereas for programs with monotonic aggregates we follow the semantics of [66].

Chapter 3

Vertex-Centric Graph Processing

3.1 Introduction

Recently, we have witnessed an explosion in size and variability of graph data. Big Graphs represent among others social, mobile, telecommunication and other networks as well as personalized health and biological data. Modern applications that manage and analyze Big Graphs pose challenges to scalability and parallelism as their computations are iterative in nature and graph data are highly interconnected. Batch-processing systems using the MapReduce programming model, although ubiquitous in applications on Big Data, suffer from performance drawbacks when used with Big Graphs.

This led to the development of large scale graph processing frameworks that specifically cater to the needs of Big Graph applications. They provide user-friendly abstractions that shield users from synchronization and communication primitives while taking care of parallelization of computation and distribution of graphs.

3.2 Vertex-Centric

Big Graph frameworks, in their majority follow the *Vertex-Centric* programming model pioneered by Pregel [48] that applied the Bulk Synchronous Parallel (BSP) model to the context of graphs. According to BSP, a graph is partitioned across nodes in a cluster and computation proceeds in a series of supersteps, each followed by a global synchronization barrier. Vertices communicate using messages, but messages sent during one superstep are visible at the destination vertex only in the next superstep.

In the *Vertex-Centric* model, a user writes a graph application from the context of a vertex by implementing a function (*vertex program*) that specifies what computation a single vertex will perform and what communication. A vertex program performs three steps: i) receive messages from neighbor vertices, ii) compute and possibly produce new values for the current vertex , and iii) send messages to neighbor vertices.

The architecture of the frameworks consists of a master node responsible for coordination, and workers that perform the actual computation. The input graph is partitioned across workers usually using a hash-partitioning scheme. A worker node may host multiple partitions that compute in parallel. The master terminates computation when there are no more messages in the system.

Pregel inspired the development of several other systems. Apache Giraph [1] is the open-source alternative to Pregel built on top of Hadoop. Giraph runs graph processing jobs as map-only jobs on Hadoop and uses HDFS for data input and output. In this thesis, we based our implementation and experiments on Giraph due to its maturity and wide-spread use in academia [35] and industry [17]. GPS [61] addresses large-degree vertices in powerlaw graphs by partitioning their edges and creating mirrors of the vertices in those partitions. Communication overhead is reduced as high-degree vertices first send messages to their mirrors, sending one message per partition. Mizan [42] suggests the usage of dynamic repartition to address the problem of load imbalance and stragglers. Practice has showed however, that the benefits gained do not justify the overhead of dynamic repartitioning.

GraphLab [46], supports both synchronous and asynchronous computation via a shared memory interface where vertices can directly read data of their neighboring vertices without the use of messages. PowerGraph [30] addresses powerlaw graphs by following a vertex-cut graph partitioning mechanism whereby edges are not allowed to cross partitions.

This is fundamentally different from the approaches based on Pregel that may suffer from load imbalances and stragglers that slow down the entire computation.

Besides the above dedicated Big Graph frameworks, there have been approaches to extend general batch-processing systems with graph computation capabilities. In the beginning, Twister [23], HaLoop [15], PrIter [81] focused on Hadoop extensions with loop-aware scheduling, in-memory caching and prioritized execution. Later, Spark [76], Naiad [52] and Hyracks [11] extend MapReduce with general purpose data operators and iterative computation as well specialized libraries for graphs. For example, GraphX [31] is a library built on top of Spark that represents graphs as collections and expresses graph computation using general data parallel primitives. They implement a host of optimizations to achieve performance comparable to bespoke graph processing engines. Of notable mention is Pregelix [14] that maps graph analytics to logical plans evaluated using database query evaluation techniques. The front-end is *Vertex-Centric* but the evaluation differs greatly from all aforementioned systems as graphs are modeled relationally and computation is achieved via physical relational operators such as outerjoins, groupby, etc,.

3.3 Think like a graph

The *Vertex-Centric* paradigm, although more user-friendly, incurs large synchronization and communication overheads. A different approach is followed by [70, 67] that suggest a subgraph-centric programming model where users express their algorithms with sub-graphs as first class citizens. The Subgraph-Centric model offers users greater flexibility as it allows low-level access to a partition of vertices as a whole as well as their messages. This way, they can exploit characteristics of their algorithms and use enhanced data structures or local asynchrony to offer greater performance. In fact, experiments reveal orders of magnitude better performance than the *Vertex-Centric* model, especially when combined with a smart partitioning algorithm. The downside of Subgraph-Centric model is the increased complexity of the programs. We will see in Chapter 4 how we take advantage of these findings without changing the *Vertex-Centric* API users express their graph analytics.

Chapter 4

Declarative Graph Analytics on Vertex Centric Engines

4.1 Introduction

Datalog has seen a recent resurgence in a wide area of applications including declarative networking [45], network provenance [82], distributed programming [6] and program analysis [12]. The desirable declarative and concise specification of graph queries and recursive evaluation semantics of Datalog make it a perfect candidate for Big Graph analytics.

[62, 71] built bespoke distributed datalog evaluation engines that however, do not fit in Big Data workflows. Graph analytics, more often than not, are part of a workflow that integrates different systems and varying data sources. Another approach, followed by [65] is to extend existing dataflow systems. They involve significant code changes to the underlying system to support efficient recursion and hence their solutions are not applicable to other systems. None of the aforementioned approaches addresses declarative graph analytics on graph processing systems that naturally support recursion.

In this chapter, we utilize Datalog to author graph analytics for execution on *Vertex-Centric* systems. We align Datalog's fixpoint evaluation semantics where rules evaluate until they generate no new facts, with the *Vertex-Centric* computation paradigm, where vertices execute until they receive no more messages.

Our approach does not require any modifications to the underlying engine to support efficient iterative evaluation. Moreover, it does not require changes to the Datalog syntax to conform to the *Vertex-Centric* paradigm. Indeed, queries in our work are expressed in the same manner as with prior work.

To this end, we employ novel query rewriting techniques to transform general Datalog queries to equivalent ones that are aware of distribution and conform to the *Vertex-Centric* model. Moreover, we employ logical optimizations that reduce computation and communication overheads. The benefit of our approach is that the optimizations are implemented as query rewriting rules making Datalog compilation and evaluation portable to other *Vertex-Centric* systems.

In this chapter we present DATALOGRAPHY, a declarative framework for *Vertex-Centric* systems and make the following contributions:

- We define VC-Datalog, an extension of Datalog that incorporates distribution and conforms to the *Vertex-Centric* paradigm.
- We design and implement a compiler that rewrites general Datalog queries to efficient VC-Datalog queries.
- We present a logical abstraction, namely *super-vertices*, that enables set-at-a-time evaluation without changing the user API or the underlying system.
- Super-vertices are the foundation for incorporating optimizations used in database and batch-processing research. Here, we show how we can marry eager-aggregation with map-side combiners to reduce communication overheads.
- We present experiments for three graph queries on three real-world graphs and find that DATALOGRAPHY offers a speedup of 2.3x-7.8x over imperative graph analytic on Giraph whereas it achieves a speed-up of 1.4x-2.5x when compared to a custom partition-aware implementation of Giraph.

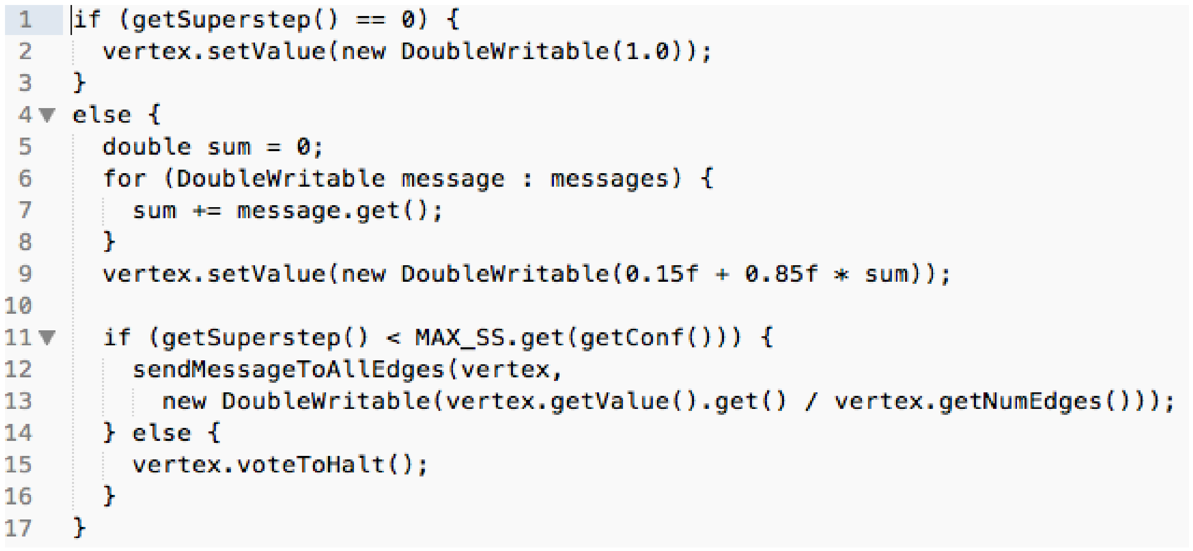
Outline: The rest of the Chapter is organized as follows. In Section 4.2 we compare the imperative implementation of a graph analytic versus its declarative one. Section 4.3 reviews support for aggregates in recursive Datalog queries. Section 4.4 presents the compiler and the query rewriting algorithms. Section 4.5 describes the logical optimizations. Section 4.7 presents experimental results including the effect our optimizations have on runtime and communication. Section 4.8 reviews related work and Section 4.9 concludes this chapter.

4.2 Declarative Big Graph analytics

We begin this section by comparing the imperative implementation of a graph analytic in Giraph versus its declarative counterpart. We see that DATALOGRAPHY, besides offering a concise language, enables developers to express their graph analytics without worrying about distribution or the underlying computation model. Indeed, queries in DATALOGRAPHY are expressed in the same manner as prior work where evaluation happens on bespoke or batch-processing engines.

Developers evaluate graph analytics on *Vertex-Centric* engines by implementing a vertex program, a function specifying the computation and communication every vertex performs. As an example, the Giraph program in Figure 4.16 computes PageRank. In superstep 0 every vertex initializes its value (rank) to 1. Every subsequent superstep involves the following steps: Line 7 sums all the messages received from neighbors. Line 9 computed the new rank using this sum and updates the vertex value. Line 11 checks if the current superstep is smaller than the max allowed superstep (specified by the user). If so, Lines 12-13 send a message to all outgoing neighbors with the quotient of the new rank and the out-degree of a vertex. If the maximum number of supersteps has been reached, Line 15 halts the computation and no messages are sent.

On the other hand, in DATALOGRAPHY, a developer uses Query 5, with no special consideration of the *Vertex-Centric* model. Reading the query, the first rule computes


```java
if (getSuperstep() == 0) {
  vertex.setValue(new DoubleWritable(1.0));
}
else {
  double sum = 0;
  for (DoubleWritable message : messages) {
    sum += message.get();
  }
  vertex.setValue(new DoubleWritable(0.15f + 0.85f * sum));

  if (getSuperstep() < MAX_SS.get(getConf())) {
    sendMessageToAllEdges(vertex,
      new DoubleWritable(vertex.getValue().get() / vertex.getNumEdges()));
  } else {
    vertex.voteToHalt();
  }
}
```

Figure 4.1. Running example: Imperative implementation of PageRank on Giraph

the out-degree of a vertex by counting its outgoing edges. The second rule, `out-rank` represents the portion of its rank a vertex sends to its outgoing neighbors. The third rule initializes the rank of every vertex at superstep 0 to 1. The last rule, computes the rank of a vertex by summing the contributions from incoming links. Note how the aggregate does not depend on the previous rank of a vertex.

out-degree$(x, COUNT(y)) \leftarrow$ edges(x,y).
out-rank$(x, r/d) \leftarrow$ pagerank(x,r), out-degree(x,d).
pagerank$(x,1) \leftarrow$ superstep$(x,i), i == 0$.
pagerank$(x, SUM(r)) \leftarrow$ edges(y,x), out-rank(y,r), superstep$(x,i), i < \sigma$.

Query 5. Running example: Declarative specification of PageRank using Datalog

4.3 Efficient recursive aggregation

Aggregation is fundamental to many graph algorithms – many compute until an aggregate value, such as MIN, MAX, SUM, or COUNT reaches a certain threshold. We discussed in Chapter 2 how the research community has yet to reach a consensus on how to support aggregation in recursion in an efficient manner. In fact, each of the prior work on Datalog evaluation on large-scale engines supports aggregates differently. Although they agree on the monotonicity of MIN and MAX aggregates, the story changes for SUM and COUNT that in general are not monotonic.

[62] supports monotonic MIN/MAX through the use of partial orders and lattices whereas SUM/COUNT that are considered non-monotonic are supported through imperative stratification. [71] also defines monotonicity using partial orders on the domain of aggregate functions but consider SUM/COUNT to be *bag monotonic*. We discussed in Chapter 2 how approaches on partial orders suffer from difficulties in proving correctness of semantics. [66], on the other hand, supports monotonicity through *continuous aggregates* where MIN,MAX,SUM,COUNT are monotonic as long as their values when viewed as a sequence are monotonic.

In this work, we follow the semantics of [66] for monotonic aggregates. However, many important algorithms require non-monotonic aggregates, such as PageRank whose summation is not monotonic: when the rank of a vertex is viewed as a sequence across all iterations, it is not monotonically increasing or decreasing. We support such queries using XY-stratification [77] but with a twist.

Explicit stratification, in general, results in awkward query formulations and slow evaluation when compared to queries whose aggregate is pushed into recursion (see Chapter 2). But we identify a subclass of queries for which explicit stratification gives both concise queries and no performance penalties. The key idea is whether aggregation depends on its previous values or not. For example in PageRank, the rank of a vertex does not depend on its own previous rank. Compare this to SSSP, where for a path to be considered the shortest, it must be compared against all previous paths of that vertex.

Using explicit stratification, PageRank is expressed like:

$\textbf{out-degree}(x, COUNT(y), 0) \leftarrow \text{edges}(x, y).$
$\textbf{out-rank}(x, r/d, i) \leftarrow \text{pagerank}(x, r, i), \text{out-degree}(x, d, 0).$
$\textbf{pagerank}(x, 1, 0).$
$\textbf{pagerank}(x, SUM(r), i+1) \leftarrow \text{edges}(y, x), \text{out-rank}(y, r, i), i < \sigma.$

Query 6. PageRank: Explicit stratification for non-monotonic aggregation

4.4 VC-Datalog

Although developers need not know, their queries are evaluated on a distributed *Vertex-Centric* engine meaning the EDB, IDB relations of their queries are distributed. We define *VC-Datalog* that extends Datalog with distribution primitives and whose rules conform to the *Vertex-Centric* paradigm.

4.4.1 Vertex-Centric normal form

We introduce the notion of *locality* for tables, specified by the first term in a predicate (variable x) called the *location specifier*. If the predicate is the head of a rule, variable x denotes the vertex evaluating the rule and the location where the resulting table is stored. If the predicate is in the body of a rule, it denotes the location of the input table. Therefore, predicates in the body of a rule that have a different first variable than that of the head are *remote* tables that neighboring vertices must sent to the currently executing vertex as messages. Note, Datalog queries must be guarded with respect to the location specifiers, i.e., the first variable of a predicate cannot be free but must be guarded by `edges` predicates if it differs from the location specifier of the head.

Using the location specifiers, we define a normal form for VC-Datalog rules where a rule evaluated at a vertex x refers to remote predicates that are incoming/outgoing neighbors of it.

Definition 1 (Vertex-Centric)**.**

$$\begin{aligned} r(x,\bar{v}) \vdash & P_x(x,\bar{\chi}), && (4.1)\\ & edges(\bar{y},x), && (4.2)\\ & P_y(\bar{y},\bar{\psi}), && (4.3)\\ & edges(x,\bar{z}), && (4.4)\\ & P_z(\bar{z},\bar{\zeta}), && (4.5)\\ & && (4.6) \end{aligned}$$

such that $\bar{v} \in \bar{y} \cup \bar{z} \cup \bar{\chi} \cup \bar{\psi} \cup \bar{\zeta}$.

according to which a rule evaluated at a vertex x has access to its local predicates P_x (line 1), its incoming neighbors (line 2) and their predicates P_y (line 3), its outgoing neighbors (line 4) and their predicates P_z (line 5).

Besides ensuring that a query can be evaluated on a *Vertex-Centric* engine, the normal form also guarantees efficient evaluation of it. As an example, consider Query 7 that finds all the paths of length 3. Rule **3-hop** has two remote predicates, the **edges** predicate of a vertex y that is an outgoing neighbor of x and the **edges** predicate of z that is an outgoing neighbor of y. Vertex x cannot evaluate this query in one superstep as remote predicate $\mathbf{edges(z,w)}$ requires two supersteps to reach it. Hence, x is not evaluating anything on the first superstep but rather needs to wait until it receives all remote relations. In general, a rule containing a path of `edges` predicates of length $n > 1$ from the location specifier requires $n-1$ supersteps for its evaluation.

3-hop$(x,y,z,w) \leftarrow$ edges(x,y), edges(y,z), edges(z,w)

Query 7. Paths of length 3

Datalog queries that don't conform with the normal form are rewritten using the normalization algorithm described below. Then Query 7 is rewritten into Query 8.

Table 4.1. Notations used

Notation	**Meaning**
R	Rule $R(x,\bar{v}) \rightarrow P_1, \ldots P_n$
$loc(R)$	Variable x that is the location specifier of R
$head(R)$	Head of rule: $R(x,\bar{v})$. Defines schema of rule output.
$body(R)$	Body of rule: $P_1, \ldots P_n$. Conjunction of predicates in rule definition
$pred(P_i)$	Predicate $P_i \in body(R)$.
$vars(A)$	Set of variables appearing in A
$\mathbb{V}(T)$	Set of vertices in T
$T_{\downarrow x}$	Subtree rooted at vertex x
$A; x$	Add element x to set A
$\mathcal{P}_{x_i}$	Predicate list of vertex x_i
$\mathcal{P}_{(x_i,x_j)}$	Predicate list of edge (x_i, x_j)

Evaluation now requires one superstep, all neighbors of x send their **1-hop** relation to it, and then x can evaluate rule **3-hop**.

1-hop$(x,y) \leftarrow$ edges(x,y)
3-hop$(x,y,x) \leftarrow$ 1-hop(y,z), edges(x,y)

Query 8. Normalized Paths of length 3

4.4.2 Vertex-Centric rewriting

The normal form ensures that every rule in a Datalog query uses **edges** predicates whose one variable is x, the location specifier of the head. As we saw though, in general this is not the case. Here, we define a rewriting algorithm that rewrites a non-normal rule into multiple rules, each of which is normal.

For reference, the notations used through the remainder of this chapter are listed in Table 4.1.

The main idea of the rewriting is that the body of a rule can be viewed as a distributed join, where IDB predicates are the distributed tables, their site denoted by their location specifiers, and the EDB predicates **edges** are the communication links connecting the remote sites. The site holding the result of the distributed join is the location specifier of the head of the rule.

As an example, consider Query 9 comprised of one rule R, that does not follow our normal form as it refers in its body to neighbors that are more than one hop away. $body(R)$ contains the local IDB predicate $\mathbf{P}$ and the remote IDB predicates $\mathbf{P_2}$ residing at vertex x_2 and $\mathbf{P_6}$ residing at vertex x_6. Moreover, it contains local and remote EDB **edges** predicates residing at their location specifiers.

$$\boxed{\begin{array}{l}\mathbf{R}(x,\ldots) \leftarrow \mathrm{edges}(x,x_1),\mathrm{edges}(x_1,x_3),\mathrm{edges}(x_3,x_4),\mathrm{edges}(x_4,x_1),\mathrm{edges}(x_1,x_2),\\ \quad \mathrm{edges}(x,x_5),\mathrm{edges}(x_5,x_6),\mathrm{P}(x,x_1,x_2),\mathrm{P}_2(x_2,x_3,x_4),\mathrm{P}_6(x_6,x_2)\end{array}}$$

Query 9. Normalization example

Evaluating R involves joining the local and remote relations to find an assignment for the variables that will make its body true. One way to achieve this is for every vertex to send its relations to x, the vertex evaluating R. This means, vertices $x_1 - x_6$ send their relations to x. One can easily see that this evaluation strategy is not optimal: 1) Communication requires multiple supersteps as in the *Vertex-Centric* paradigm, a vertex can send messages only to its direct neighbors. For example, relation $\mathbf{edges(x_3,x_4)}$ requires 2 supersteps to reach x, one where x_3 sends the relation to x_1 and then x_1 sends it to x. 2) Moreover, there are multiple paths along which relations can be transmitted. Instead of following the previous path, x_3 can send its relation to to x_4, x_4 to x_1 and then x_1 sends it to x which requires 3 supersteps. 3) Finally, remote relations are transmitted in their entirety over a series of vertices when few tuples are actually relevant. For instance, relation $\mathbf{P_2(x_2,x_3,x_4)}$ is sent from x_2 to x_1 and then to x.

However, every vertex appearing in the body of a rule is an evaluation site. Hence, we can distribute computation among these vertices where each evaluates parts of the rule that are local to it. The added benefit is that only relevant information is transmitted further along the route to x since vertices behave as filters.

The problem of optimizing distributed queries has been studied extensively in the context of distributed databases [43]. The most expensive operations are joins and much works focused on optimizing them by reducing transfer costs between sites (semi-join, join reductions) and increasing parallelism of query evaluation (query and data shipping). In this work, in the absence of statistics, we propose a greedy solution that minimizes the total number of supersteps and the size of relations transmitted while following the constraints imposed by the *Vertex-Centric* paradigm.

Our rewriting transforms an initial Datalog rule to a collection of VC-Datalog rules by assigning location to distributed joins and deciding along which links to send messages. The rewriting guarantees that in the new rules: i) Communication occurs only between vertices x_i,x_j that are direct neighbors, i.e, $\mathbf{edge(x_i, x_j)} \in$ input graph. ii) A vertex evaluates a join within one superstep, having received all remote predicates in the preceding superstep. ii) The site where the result of the join will reside is the location specifier of the rule of the initial rule.

The rewriting consists of the three steps shown in Figure 4.2. First, we create a *Routing-Tree*, that depicts how vertices will transmit remote predicates so that the final results of query evaluation reside at the location specifier of the initial rule. Next, we annotate the nodes and edges of the Routing-Tree with IDB predicates, essentially assigning the evaluation site to distributed joins. Finally, we create a new rule for every node in the Routing-Tree that is annotated with predicates. The result of the rewriting is a set of rules that follow the *Vertex-Centric* normal form.

Create Routing-Tree

The **edges** predicates in the body of a rule specify a graph where for every predicate $\mathbf{edges(x_i, x_j)}$ we create two nodes x_i, x_j and connect them via an edge. We refer to this graph as the *Body-Graph* and Figure 4.3 shows the Body-Graph for rule R.

The Body-Graph represents the allowed channels of communication: which vertex can send messages to what other vertex, since in *Vertex-Centric* they must be connected

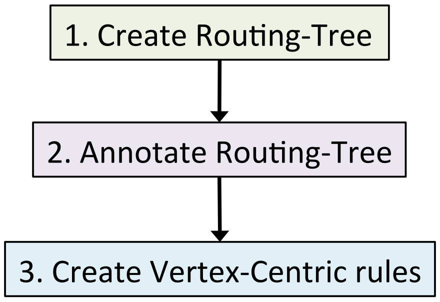


Figure 4.2. Steps followed to transform a general Datalog query into a *Vertex-Centric* query

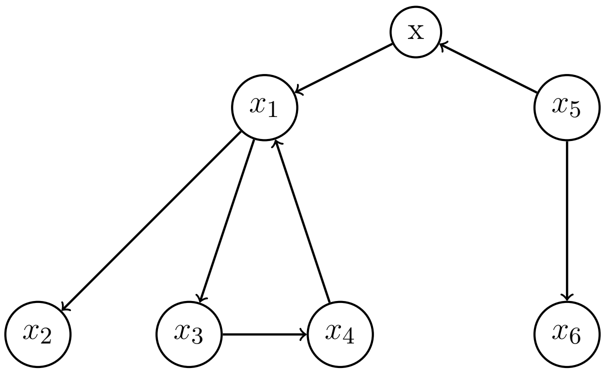


Figure 4.3. The graph representation of *body*(R).

through a direct edge. There are multiple ways for two vertices to communicate between each other: For example, vertices x_3 and x_4 share the edge (x_3, x_4) but they are also connected through the path $(x_4 \rightarrow x_1 \rightarrow x_3)$.

Our minimization objective is $R's$ evaluation to require the least number of super-steps satisfying the constraints imposed by *Vertex-Centric* and the site holding the result (vertex x). Thus, we create the Routing-Tree for R ($T_{route}(R)$) in Figure 4.4 through a BFS traversal of the Body-Graph with edge directions such that all paths lead to x (x is the root of the tree). The Routing-Tree is guaranteed to have the smallest height.

Annotate Routing-Tree

The next step in the rewriting is to decide the evaluation site for every predicate. We identify two classes of predicates: the *neighbor-join* have arguments that are all from a single vertex or vertices that are neighbors in $T_{route}(R)$. On the other hand, the *path-join*

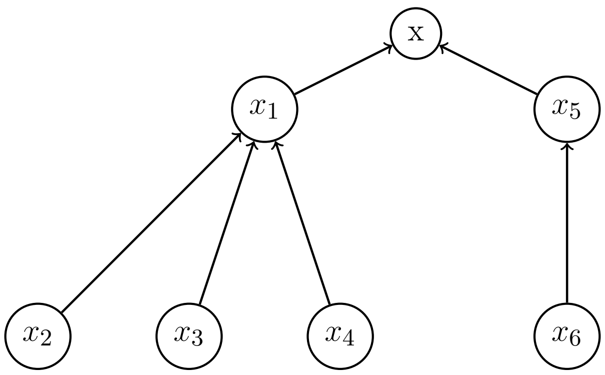


Figure 4.4. The Routing-Tree of R denoting how vertices will communicate during query evaluation.

predicates have arguments that are connected through a path of length > 1. For example, a predicate $P_3(x_3, x_4)$ would be a neighbor-join as x_3 and x_4 are neighbors. On the other hand, P_2 is a path-join predicate as its arguments x_2,x_3 and x_4 are not connected through a direct edge. The same holds for predicates P and P_6.

Neighbor-join predicates are assigned to their location specifier vertex. For example, $\mathbf{edges(x_3, x_4)}$ is assigned to vertex x_3 and requires no communication for its evaluation. In general, neighbor-joins are evaluated in one superstep, requiring either no communication or messages from their neighbors. Path-join predicates are placed on the vertex that is the least common ancestor (lca) of their arguments. For example, $\mathbf{P_2}$ is assigned to vertex x_1 that is the lowest common ancestor of its arguments. Although $\mathbf{P_2}$ can be evaluated in one superstep, with vertices x_2,x_3 and x_4 sending their relations to x_1, this is generally not true for path-join predicates. Consider predicate $\mathbf{P_6}$ whose arguments are x_6 and x_2. Their lca is vertex x to which $\mathbf{P_6}$ is assigned to.

Choosing the least common ancestor as the evaluation site satisfies our minimization objective as the lca is the meeting point for remote vertices on their shortest paths to the root. Choosing either x_2 or x_6 as the evaluation site for $\mathbf{P_6}$, would require 4 supersteps whereas choosing x requires 2 supersteps. Moreover, choosing the lca satisfies the constraint of the final result of evaluation being on x, the location specifier of the initial rule.

The algorithm for annotating the Routing-Tree is in Algorithm 2. It traverses the Routing-Tree bottom-up and left-to-right and adds annotations to its nodes and edges that are a list of predicates $\mathcal{P}$. For the nodes of tree the predicate list denotes the predicates that can be evaluated at that vertex. The predicates on the edges denote the messages that must be sent between vertices and what those messages should contain.

For every vertex x_i, add predicate P_i to its list of predicates $\mathcal{P}_i$ if: i) All variables in P_i are x_i or appear in an edge predicate with x_i its first variable (lines 5,6). ii) All variables in P_i appear in the subtree rooted at x_i ($T_{\downarrow x_i}$) (lines 11,12). Else, add P_i to the predicates list of edge $(x_i, x_j) \in T_{route}$ where x_j is parent of x_i (lines 14,15).

Algorithm 2. Edge-Tree annotation algorithm

```
1: function ANNOTATE($R(x, \ldots)$)
2:     $T_{route}(R) \leftarrow$Routing-Tree of $R$ rooted at $x$
3:     for $x_i \in$ bottom-up, left-to-right traversal of $T_{route}(R)$ do
4:         for $pred(P_j) \in body(R)$ do
5:             if  $P_j$ is $\mathbf{edges(x_i, x_k)}$ then
6:                 $\mathcal{P}_{x_i}; P_j$
7:             end if
8:             if $loc(P_j) = x_i$ then
9:                 Let $T_{\downarrow x_i}$ subtree of $T_{route}(R)$ rooted at $x_i$
10:                Let $\mathbb{V}(T_{\downarrow x_i})$ set of vertices in $T_{\downarrow x_i}$
11:                if $vars(IDB(P_j, R)) \subseteq \mathbb{V}(T_{\downarrow x_i})$ then
12:                    $\mathcal{P}_{x_i}; P_j$
13:                else
14:                    Let $x_a$ parent of $x_i$ and edge $(x_i, x_a) \in T_{route}(R)$
15:                    $\mathcal{P}_{(x_i, x_a)}; P_j$
16:                end if
17:            end if
18:        end for
19:    end for
20: end function
```

The result of Algorithm 2 is shown in Figure 4.5. We see that every predicate in the body of R is assigned to a vertex, once. Moreover, the invariant at every step of the algorithm is that whenever a predicate P_i is assigned to a vertex x_i all variables in P_i appear in the subtree rooted at x_i and are available in its children or itself.

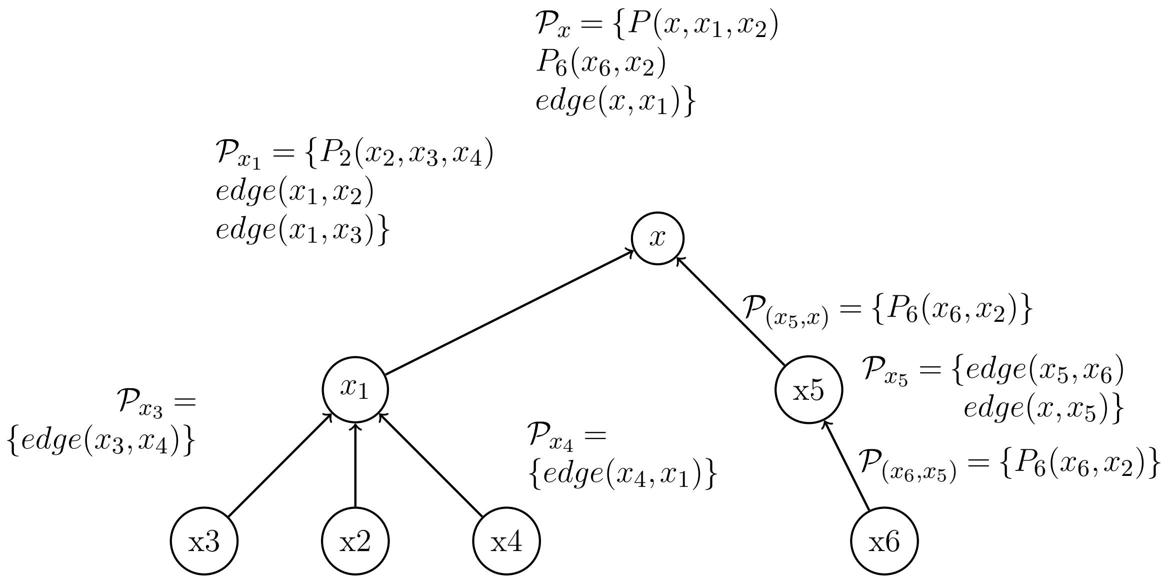

$\mathcal{P}_x = \{P(x,x_1,x_2)$
$P_6(x_6,x_2)$
$edge(x,x_1)\}$
$\mathcal{P}_{x_1} = \{P_2(x_2,x_3,x_4)$
$edge(x_1,x_2)$
$edge(x_1,x_3)\}$
x
$\mathcal{P}_{(x_5,x)} = \{P_6(x_6,x_2)\}$
x_1
x5
$\mathcal{P}_{x_5} = \{edge(x_5,x_6)$
$edge(x,x_5)\}$
$\mathcal{P}_{x_3} = \{edge(x_3,x_4)\}$
$\mathcal{P}_{x_4} = \{edge(x_4,x_1)\}$
$\mathcal{P}_{(x_6,x_5)} = \{P_6(x_6,x_2)\}$
x3
x2
x4
x6

Figure 4.5. Edge-Tree annotated with predicates annotating its vertices and edges

Create Vertex-Centric rules

The final step in the rewriting is to create the collection of rules that will replace the initial rule and that abide to the normal form.

Algorithm 3 visits the Routing-Tree bottom-up and left-to-right and creates a new rule for every vertex whose predicate list $\mathcal{P}$ is not empty. Function *Create-Rule* (line 17) takes as argument the predicate list of a vertex x_i and creates rule R_i as follows: The head of R_i has the location specifier x_i, and the rest of the arguments are all the variables appearing in the predicates of $\mathcal{P}_i$. The body of R_i is the conjunction of these predicates. Then, for every child of $x_i \in T_{route}$, if there is a rule R_j, the algorithm adds the rule to the body of R_i (line 6). The same happens for every edge $(x_j, x_i) \in T_{route}$ if it carries annotations (line 10).

The algorithm rewrites rule R of Query 9 into the rules of Query 10.

Note, Algorithm 3 includes in the head of a newly created rule R_i all variables appearing in its body irrespective of whether they are needed. We employ the Database

Algorithm 3. Rule rewriting algorithm

```
    Input $R(x,\ldots)$
    Output $R_1 \ldots R_n$
 1: function NORMALIZE(R)
 2:     Annotate($T_{route}(R)$)
 3:     for $x_i \in$ bottom-up, left-to-right traversal of $T_{route}(R)$ do
 4:         $R_i(x_i,\bar{v})$ = Create-Rule($\mathcal{P}_i$)
 5:         for edge $(x_j,x_i) \in T_{route}(R)$ do
 6:             if $\exists R_j$ then
 7:                 $body(R_i); head(R_j)$
 8:                 $\bar{v}; \{\forall v_j \in vars(head(R_j))\}$
 9:             end if
10:             if $\mathcal{P}_{(x_j,x_i)} \neq \emptyset$ then
11:                 $body(R_i); \{\forall P_j \in \mathcal{P}_{(x_j,x_i)}\}$
12:                 $\bar{v}; \{\forall v_j \in vars(P_j)\}$
13:             end if
14:         end for
15:     end for
16: end function
17: function CREATE-RULE($\mathcal{P}_{x_i}$)
18:     Create rule $R_i$
19:     $\bar{v} = \{vars(P_i) | \forall P_i \in \mathcal{P}_{x_i}\}$
20:     $head(R_i) = R_i(x_i,\bar{v})$
21:     $body(R_i)$ = Conjunction of $\forall P_i \in \mathcal{P}_{x_i}$
22:     return $R_i$
23: end function
```

$\mathbf{R_3}(x_3,x_4) \leftarrow \text{edge}(x_3,x_4).$
$\mathbf{R_4}(x_4,x_1) \leftarrow \text{edge}(x_4,x_1).$
$\mathbf{R_1}(x_1,x_2,x_3,x_4) \leftarrow \text{P}_2(x_2,x_3,x_4), \text{R}_3(x_3,x_4), \text{R}_4(x_4,x_1), \text{edge}(x_1,x_2), \text{edge}(x_1,x_3).$
$\mathbf{R_5}(x_5,x_6,x_2) \leftarrow \text{P}_6(x_6,x_2), \text{edge}(x_5,x), \text{edge}(x_5,x_6).$
$\mathbf{R}\text{'}(x) \leftarrow \text{edge}(x,x_1), \text{P}_6(x_6,x_2), \text{R}_1(x_1,x_2,x_3,x_4), \text{R}_5(x_5,x_6,x_2).$

Query 10. Result of applying Algorithm 3 to Query 9 textbook optimization of pushing projection down to truncate the heads of these rules. This results in smaller message exchange between vertices.

Theorem 1. *Let $Q = Normalize(R)$ the query created by applying Algorithm 3 on R. Then, $\forall R_i \in Q$ follows the Vertex-Centric normal form.*

Proof. Let x_i be the location specifier of R_i. From the algorithm we have that R_i has in its body predicates with location specifier x_i or x_j where edge (x_j, x_i) in $T_{route}(R)$. $\square$

Definition 2. *Let R be a rule and $P1, i\ldots, P_n$ IDB predicates in its body. We define as $\phi(R, P_1, \ldots P_n)$ the function that recursively replaces each IDB P_i in $body(R)$ with its body $body(P_i)$.*

Theorem 2. *Let $P = Normalize(R)$ the program created by applying Algorithm 3 on R. Let R' the goal of P and $R_1, \ldots, R_n$ the rest of the rules in P/R'. Then, $R \equiv \phi(R', R_1, \ldots, R_n)$.*

Proof. Rules $R_1, \ldots R_n, R'$ were created by a bottom-up traversal of $T_{route}(R)$ whose vertices and edges are annotated with all predicates of R. Since the recursive application of ϕ traverses $T_{route}(R)$ top-down, it is easy to see that the equivalence holds. $\square$

4.4.3 Compilation and Planning

A VC-Datalog query is compiled into logical plans with the help of the dependency graph (see Chapter 2) that gives an ordering of the rules. The dependency graph (G_{dep}) for PageRank (Query 5) can be seen in Figure 4.6 along with annotations of the strata for each predicate as this query must be explicitly stratified due to its non-monotonicity. We see that stratum 0 contains the initialization rules with the explicit temporal variable of 0. The next stratum has $\textbf{out-rank}^{\mathbf{i}}$ and $\textbf{pagerank}^{\mathbf{i}}$ whereas the final stratum has $\textbf{pagerank}^{\mathbf{i+1}}$. Recursion involves replacing the contents of $\textbf{pagerank}^{\mathbf{i}}$ with the contents of $\textbf{pagerank}^{\mathbf{i+1}}$.

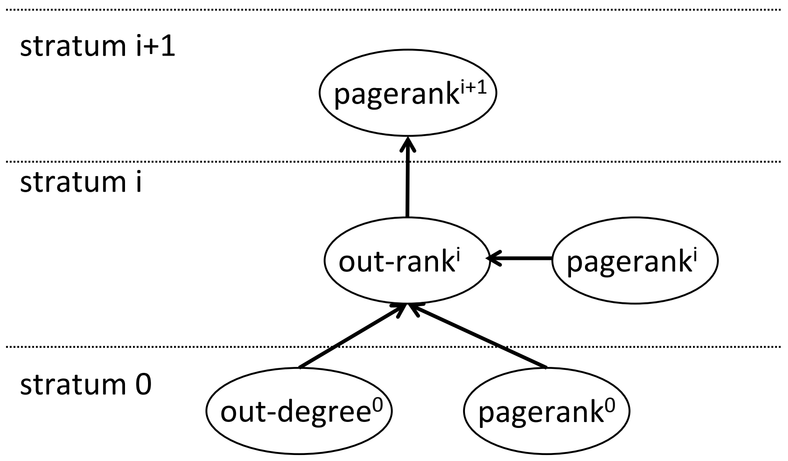
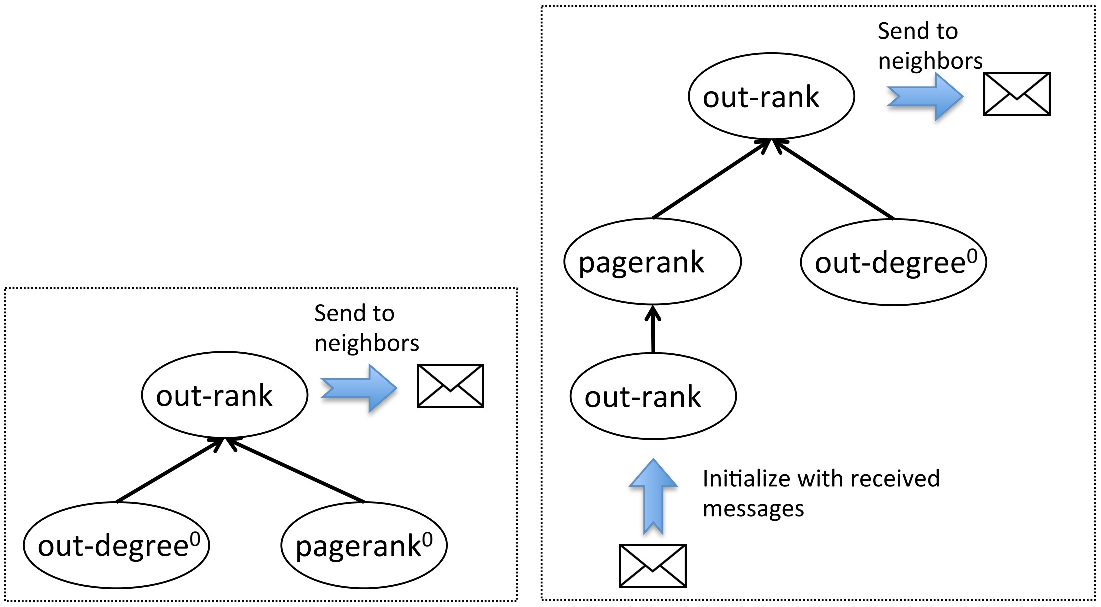


Figure 4.6. The dependency graph for running example Query 11.

Although this dependency graph is correct for a centralized setting, it does not represent how evaluation should proceed in a *Vertex-Centric* setting where relations are distributed. What we want to be represented in G_{dep} is:

- A vertex evaluates the same query at every superstep
- If there are remote predicates, they are initialized with received messages at the beginning of query evaluation in a superstep
- If there are new tuples computed for a remote predicate, they are sent to neighbors at the end of query evaluation in a superstep

Our desired query evaluation is better represented with the dependency graph of Figure 4.7 comprised of two parts: The first part, 4.7a, represents the evaluation of superstep 0 where the initialization rules get invoked to compute relation **out-rank**. Every vertex should send this relation to its neighbors. In every subsequent superstep, the second part of the dependency graph in 4.7b should be used that represents the recursive part of the query. For simplicity, we have removed the strata information from the predicates. Here, a vertex should initialize the remote predicate **out-rank** with the received messages, compute the new pagerank and then compute new tuples for its own predicate **out-rank**. If there are new tuples, it sends them to its neighbors.



(a) Initialization rules at superstep 0 **(b)** Recursive rules at superstep i

Figure 4.7. Desired dependency graph of Query 5 that captures distribution.

Our query in 5 does not convey any information on how to create the above dependency graph. That's why DATALOGRAPHY rewrites this query using the information of location specifiers to make it explicit which relations are initialized with received messages and which relations are sent to neighbors.

Algorithm 4 rewrites a vertex-centric query into an equivalent query that for every remote relation contains two special instructions: A **send-message** rule that tells DATALOGRAPHY what messages to sent to neighbors at the end of a superstep and a predicate **receive-message** that replaces the initial remote predicate and instructs DATALOGRAPHY to initialize this relation with the received message at the beginning of a superstep. Obviously, the head of rule **send-msg** must match predicate **receive-msg** as the relation that is sent as a message must be the same as the relation received.

The algorithm rewrites Query 5 into Query 11. Since there is only one remote predicate (**out-rank**), one set of messaging instructions is added. This query-level rewriting facilitates further optimizations as we will see in Section 4.5.

Algorithm 4. Message rules rewriting

```
1: $R(x,\bar{v}) \to P_1 \dots P_n$
2: for $IDB(P_i) \in body(R)$ do
3:     if $loc(R) \neq loc(P_i)$ then
4:         Create rule send-msg$_{P_i}(y, vars(P_i)) \leftarrow edges(x,y), P_i$
5:         Replace $P_i$ with receive-msg$_{P_i}(vars(P_i))$
6:         Remove from $vars(head(\text{send-msg}))$ the variables not in $vars(P_i)$
7:     end if
8: end for
```

out-degree$(x, COUNT(y), 0) \leftarrow$ edges(x,y).
out-rank$(x, r/d, i) \leftarrow$ pagerank(x,r,i), out-degree(x,d)
pagerank$(x,1,0)$.
pagerank$(x, SUM(r), i+1) \leftarrow$ edges(y,x), receive-msg(y,r,i)
send-msg$(y,r,i) \leftarrow$ edges(x,y), out-rank(x,r,i)

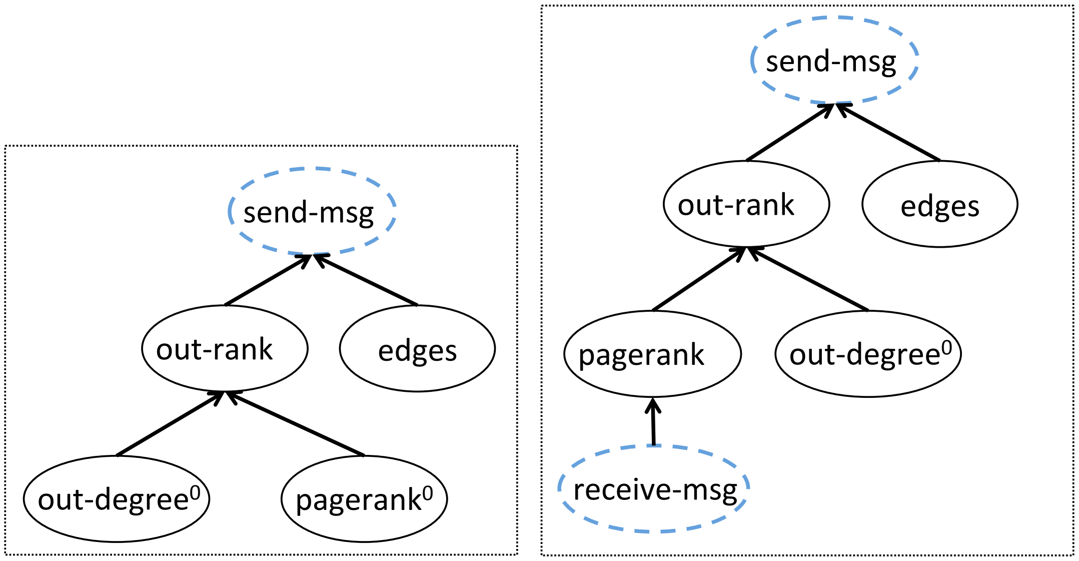
Query 11. PageRank

The dependency graph for Query 11 is in Figure 4.8. Datalography splits G_{dep} at the first **receive-message** predicate it encounters in a traversal on the topologically sorted graph.

Datalography uses the dependency graph to order the logical plans of each rule. First, the logical plan in Figure 4.9a evaluates to compute the new pagerank of a vertex using the ranks it received from its incoming neighbors. Then, the logical plan in Figure 4.9b evaluates to compute the new **out-rank**, the portions of its rank, a vertex sends to its outgoing neighbors.

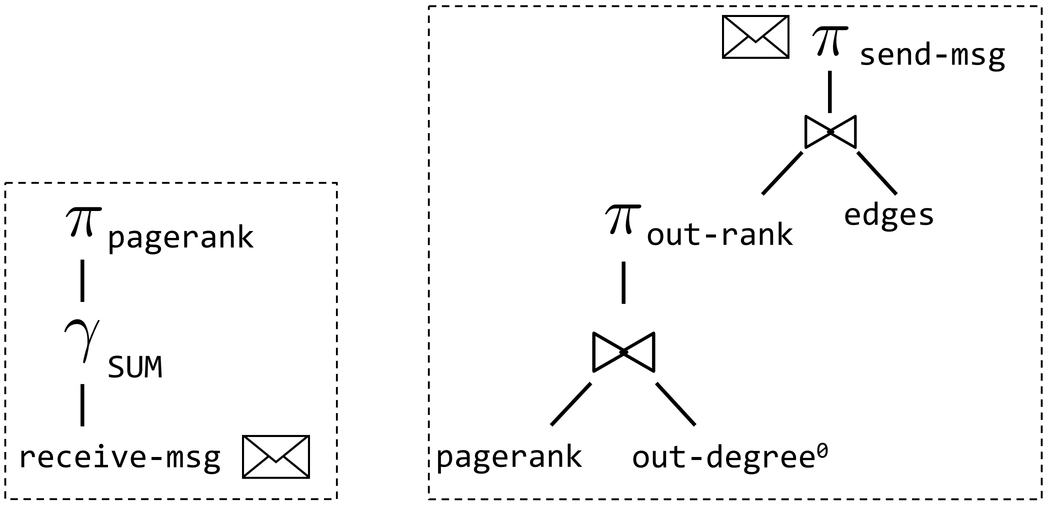
4.5 Optimizations

This section discusses optimizations developed for scalable processing of distributed graph queries. While some optimizations have been explored in the context of prior parallel data processing systems, here we investigate their novel application in a super-vertex aware Datalog evaluation engine.



(a) Initialization rules **(b)** Recursive rules

Figure 4.8. The dependency graph of Query 11.



(a) Compute new pagerank **(b)** Compute new out-rank and messages to send

Figure 4.9. The logical plan for Query 11.

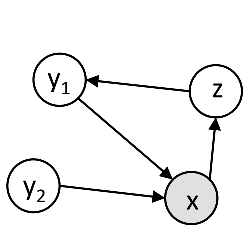
4.5.1 Compilation

Every rule is translated to a logical plan. After performing standard texbook optimizations such as pushing selection and projection down, the optimizer performs a heuristic based ordering of the joins in the plan. As such, tables received as messages are scanned first as we expect them to be small and have high selectivity whereas joins with the EDB relation `edges` are performed last.

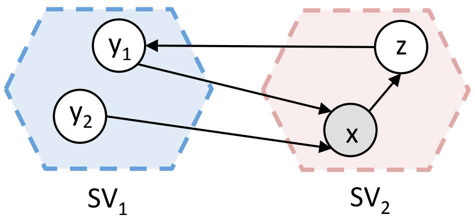
Schema of rules: In the general case, a table resulting from the evaluation of a rule has multiple rows and multiple columns but looking at the cardinality and data type of the output tables, more efficient data structure can be used. For instance, if a rule has only one variable in its head, such as $R(x)$, it is represented as a boolean table of one row as such rules simply express the existence of a tuple satisfying them. If a rule has aggregation and there is only one grouping variable such as $sssp(x, MIN(d+1)$ or $wcc(x, MIN(l))$ the cardinality of the table is again one row.

Identifying the right structure, does not only conserve memory but also reduces computation. In semi-naive evaluation, a rule is evaluated until no more new tuples can be inferred. At every step of the evaluation, the delta of the old and new tuples is added to the current rule result. Computing the delta is expensive but can be avoided for rules whose subsequent evaluation do not compute any new tuples. For instance, if the rule result is a boolean table, there is no need in computing the delta if the table already contains one tuple, as no new facts can be added to it.

Static rules: The optimizer identifies *static* rules that don't produce new tuples after their first evaluation and hence don't need to get evaluated in subsequent supersteps. For example, rule`out-degree` in Query 5 that counts the out-degree of a vertex is static. Boolean rules described above are considered static as well after after the first evaluation that populates their table.



(a) Input graph



(b) Super-vertices

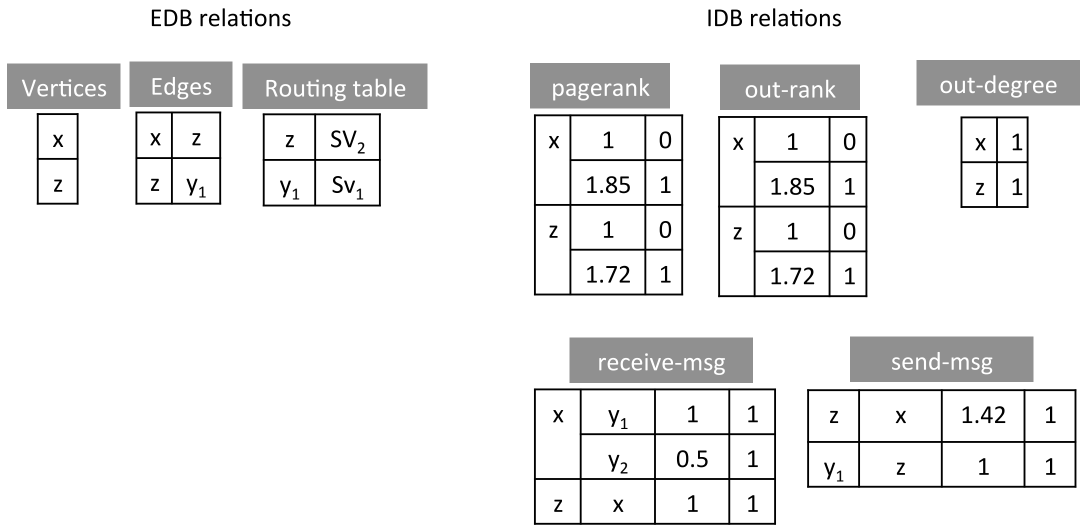
Figure 4.10. Example: Partitioning of input graph into super-vertices

4.5.2 Super-vertices

Previous work [70] extended the *Vertex-Centric* model to a Graph-Centric, where computation happens on sub-graphs instead of vertices. Their work showed that it offers greater flexibility with respect to the algorithms and computation paradigms but also better performance. However, their approach requires changes both to the user API and the underlying engine. Here, we capitalize on this idea to improve both runtime and communication overheads, in an agnostic to the developer *and* engine manner. As will see, no changes to the Datalog query or the underlying *Vertex-Centric* engine are required.

Datalography uses an abstraction that groups vertices into *super-vertices*. Consider the example input graph of Figure 4.10a and its partitioning into two super-vertices, SV_1 and SV_2 in Figure 4.10b. A super-vertex contains a collection of vertices and their edges, but edges are allowed to cross super-vertex boundaries as with the edges (y_1, x) and (y_2, x) for example. Choosing a good partitioning strategy is important as it balances the workload of partitions and reduces the number of messages transmitted over the network. Our approach however, does not depend on a particular partitioning method.

Information in super-vertices is represented in a relational format and the schema consists of the EDB relations describing the vertices and edges contained as well as the IDB relations created during query evaluation. When a vertex evaluates a query, it creates local relations. Super-vertices combine these local tables of all the vertices they contain into one large table. The same schema is the same but the multiplicity of the tables is


EDB relations
IDB relations

Vertices

x
z

Edges

x	z
z	y_1

Routing table

z	SV_2
y_1	Sv_1

pagerank

x	1	0
	1.85	1
z	1	0
	1.72	1

out-rank

x	1	0
	1.85	1
z	1	0
	1.72	1

out-degree

x	1
z	1

receive-msg

x	y_1	1	1
	y_2	0.5	1
z	x	1	1

send-msg

z	x	1.42	1
y_1	z	1	1

Figure 4.11. Schema of super-vertex SV_2

increased. For instance, super-vertex SV_2 in Figure 4.11 contains vertices x and z and its EDB relations are `Vertices`, `Edges` and `Routing table` whose role we will discuss below. The IDB relations created during evaluation of Query 5 contain the results for both x and z.

Super-vertices are an internal abstraction employed by Datalography and a developer is agnostic of them when authoring queries. For instance, no modification is needed for query 5 whether super-vertices are used or not. On the other hand, at the engine layer, Giraph is agnostic of the internal vertices and only considers super-vertices which are handled like any other regular vertex: at every superstep, a super-vertex evaluates its vertex program and sends messages to its neighboring super-vertices. Since the communication and computation layer is unaware of the internal vertices, it is the job of Datalography to evaluate a query for every internal vertex and handle the messages produced during query evaluation.

A super-vertex must gather all the messages of its internal vertices and route them to the correct neighbor super-vertex that houses the destination vertex. To do so every super-vertex maintains a routing table which maps every neighbor of an internal vertex to the super-vertex it belongs. Table `Routing table` in Figure 4.11) specifies that z, a neighbor of x is in SV_2 whereas y_1, a neighbor of z, is in SV_1. A super-vertex uses the routing table at runtime and joins it with relation `send-msg` to determine which partition of it to send to which neighbor.

4.5.3 Combiners

The abstraction of super-vertices enables set-at-a-time evaluation and allows DATALOGRAPHY to apply optimizations from database and batch-processing research to the context of graph processing.

Eager aggregation was first addressed by [74] in the context of optimization of sequential relational queries. The idea was to perform aggregations early, before joins, to minimize the input to the join operators (prior practice had been performing aggregation as the last step). We adapt eager aggregation to our context, by applying aggregation before a super-vertex sends messages and then another aggregation at the destination super-vertex effectively providing source and destination side combiners.

It is well-known that for eager aggregation to preserve the semantics of the program, an aggregation function F needs to be *decomposable*. F is decomposable if there exist aggregation functions F_1 and F_2 such that $F(S_a \uplus S_b) = F_2(F_1(S_a) \uplus F_1(S_b))$, where S_a and S_b are two bags (multi-sets) and $\uplus$ denotes bag union. It can be seen that common aggregate functions such as *sum*, *count*, *min*, and *max* are all decomposable.

When the variables of a **receive-msg** predicate are used in aggregation, DATALOGRAPHY pushes the aggregation to its corresponding **send-msg** rule when the following conditions hold:

1. Only variables of the **receive-msg** predicate are part of the aggregate variables.

2. Any variables of the **receive-msg** predicate that join with other predicates must be part of the grouping keys.

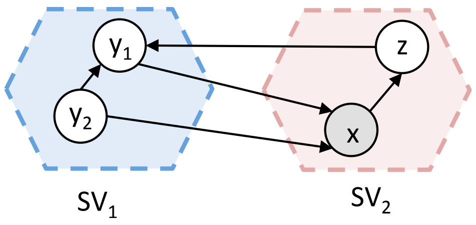
3. Predicate **receive-msg** does not appear in the body of another rule.

When the above hold, it is correct to apply source-side combiners. Query 11 is rewritten into Query 12 where the aggregate SUM is applied to **send-msg** rule. Without the abstraction of super-vertices, however, the rewriting will offer performance benefits only if a vertex sends multiple messages to the same neighbor and those messages can be aggregated together. We have not seen this scenario occur in the graph analytics we experimented with.

out-degree$(x, COUNT(y)) \leftarrow$ edges(x, y).
out-rank$(y, r/d) \leftarrow$ pagerank(x, i, r), out-degree(x, d)
pagerank$(x, 0, 1)$.
pagerank$(x, i+1, SUM(r)) \leftarrow$ receive-msg(y, r, i)
send-msg$(y, SUM(r), i) \leftarrow$ edges(x, y), out-rank(x, r, i)

Query 12. PageRank with source-side combiners

On the other hand, with super-vertices, the rewriting offers performance benefits if multiple internal vertices in a super-vertex send messages to the same destination vertex. Super-vertices gather the messages their internal vertices send in relation **send-message**. For example, we see that in Figure 4.12a super-vertex SV_1 has three tuples in relation **send-msg**, two of which are tuples destined to vertex x in SV_2. Mirroring this, relation **receive-msg** of super-vertex SV_2 has three tuples, two received by vertex x. When we apply the optimization of eager-aggregation, the number of tuples in these relations is reduced by aggregating the tuples destined to x. We see that in Figure 4.12b relation **send-msg** has now two tuples, and the tuple destined to x has the sum of the **out-rank** of y_1 and y_2. In our experiments (Section 4.7), we see that this optimization offers a $3x - 6x$ speedup.



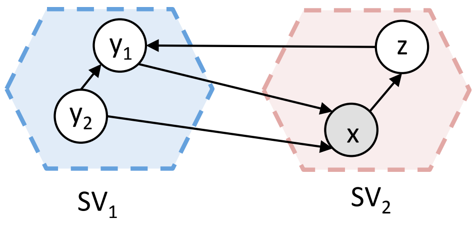
SV_1: send-msg

x	y_1	1	1
x	y_2	0.5	1
y_1	y_2	0.5	1

SV_2: receive-msg

x	y_1	1	1
x	y_2	0.5	1
z	x	1	1

(a) Send messages without source-side combiners



SV_1: send-msg

x	1.5	1
y_1	0.5	1

SV_2: receive-msg

x	0.5	1
z	1	1

(b) Send messages with source-side combiners

Figure 4.12. Messages sent with eager-aggregation

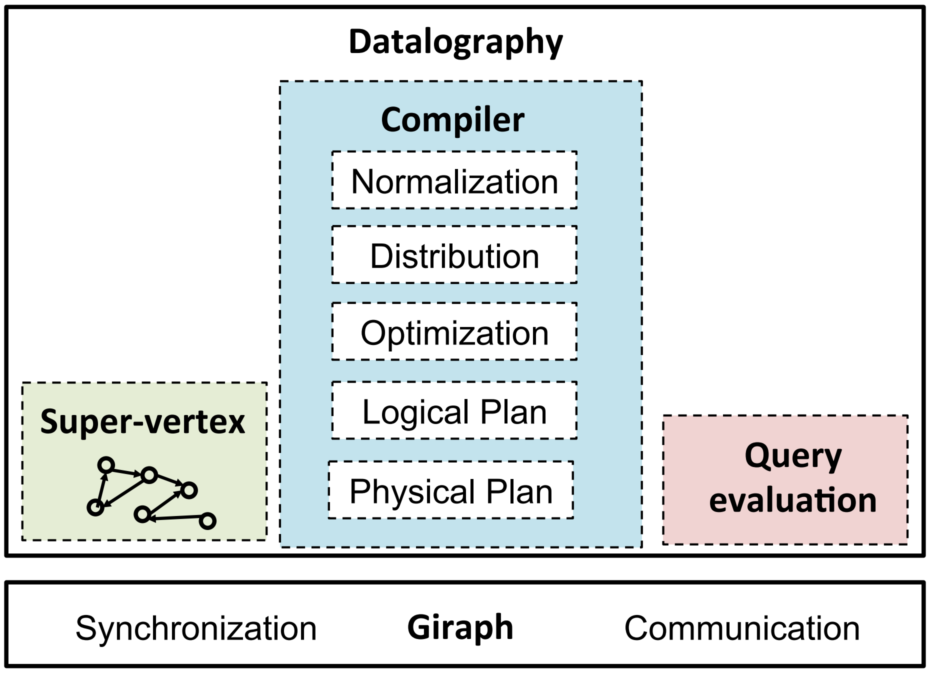


Figure 4.13. Datalography architecture

4.6 Implementation

Datalography is built on the analytic layer of Giraph as seen in Figure 4.13, the same way a graph analytic like PageRank would be implemented. As such, the query engine is a vertex program that implements the abstraction of super-vertices, compiles a general Datalog query into an optimized *Vertex-Centric* physical plan and evaluates the query for all internal vertices. Messages created during query evaluation are forwarded to Giraph that takes care of communication and synchronization between super-vertices.

Besides an API for the vertex program, Giraph provides an API for extending the worker and master compute programs. Datalography implements the worker compute program to create the dependence graph at the beginning of computation and the logical plans for every rule. Then, at the beginning of every superstep, the workers select the next plans to evaluate in a coordinated manner: every worker and every super-vertex evaluate the same rules in a superstep. At the end of a superstep, super-vertices notify the master whether new tuples were created during query evaluation indicating that computation should continue. By default, the master only checks for the existence of new messages in determining termination so Datalography needs to refine this condition to take into account tuple creation. When convergence is achieved and no more new tuples are created (and hence, no new messages) by any super-vertex, the master halts the computation.

4.7 Experimental evaluation

We compare the performance of implementing graph algorithms in Datalog and executing them in Datalography, versus implementing them in customized Java code executed directly in Giraph. We choose algorithm implementations shipped with the Giraph example package. The comparison shows that running Datalog implementations of graph algorithms on Datalography offers a speedup of 2.3x-7.8x (depending on the dataset and algorithm) over Giraph whereas it achieves a speed-up of 1.4x-2.5x when compared to a custom partition-aware implementation of Giraph (Giraph-Metis). Moreover, we investigate how super-vertices and eager-aggregation optimizations employed by Datalography affect runtime.

4.7.1 Experimental setup

All experiments were run on a cluster of 8 nodes, each with 32GB of RAM, 4 vCPUs and 1Gb/s NICs. The nodes are running Ubuntu 14.04, with jdk-1.0.7. One of the nodes was designated as the Hadoop master and did not participate in computations. We installed Hadoop 2.5 and Giraph 1.2

We used real-world datasets[1], namely indochina-2004 (IN-04), uk-2002 (UK-02), arabic (AR-05) and uk-2005 (UK-05). Their characteristics can be seen in Table 4.2.

For the assignment of vertices to super vertices we used the ParMetis [3] graph partitioning library. Section 4.5 made the case for super-vertices as they enable a more sophisticated partitioning of the vertices: every super vertex contains vertices of the initial

[1]http://www.dis.uniroma1.it/challenge9/download.shtml

Table 4.2. Dataset characteristics

Dataset	V	E	Avg Degree	Avg Diameter
IN-04	7.4M	194M	26.17	28.12
UK-02	18.5M	298M	16.01	21.59
AR-05	22.7M	640M	28.14	22.39
UK-05	39.5M	936M	23.73	23.19

input graph that are "close" to each other to minimize communication between different super-vertices.

Graph analytics

We consider three different algorithms, SSSP, WCC and PageRank. In the experiments below we compare the running time when executing their customized Java implementation on Giraph and Giraph-Metis versus executing the Datalog implementation on Datalography.

Single Source Shortest Paths computes the distance between a single source and all other vertices in its connected component. The Datalog query that evaluates the same algorithm is shown in Query 13.

We compare against the implementation provided by Giraph in its examples package. The source from which the algorithm starts initializes its distance with 0. A vertex receives the distance from all its neighbors and picks the minimum. If the minimum is smaller than the current distance, it propagates it to its outgoing neighbors. Note that we assume all edges have unit weights. In SSSP, in the first superstep all vertices are inactive except the source. A vertex gets woken up when it receives a message. The compute invocations and hence network usage starts out small, peaks and then reduces again. The algorithm runs for a maximum number of supersteps equal to the diameter of the graph.

In our experiments the source for IN-04 and AR-05 is the vertex with ID=10000 whereas for UK-02 and UK-05 it is the vertex with ID=0. SSSP requires 43 supersteps for

IN-04, 47 supersteps for UK-02, 57 supersteps for AR-05 and 201 supersteps for UK-05.

$$\begin{array}{l}\textbf{sssp}(x,0) \leftarrow \text{vertices}(x), x == \alpha. \\ \textbf{sssp}(x, \text{MIN}(d+1)) \leftarrow \text{edges}(y,x), \text{sssp}(y,d).\end{array}$$

Query 13. SSSP

Weakly Connected Components finds components in a graph, that is subgraphs in which there is a path from every vertex to every other one, ignoring edge direction. We used the implementation of HCC algorithm provided by Giraph's example package in which the component for all vertices is initialized to the vertex ID. At every superstep, a vertex sends it's component ID to its outgoing neighbors. The receiving vertex compares the component ID's it has received to it's current one and keeps the smallest. If the component ID of a vertex changed due it receiving a smaller one, the vertex forwards the new one to its neighbors.

Unlike SSSP, in WCC all vertices are active initially. The compute invocations drop as the computation proceeds and the supersteps are bounded by the diameter of the graph. WCC requires 39 supersteps for IN-04, 116 supersteps for UK-02, 51 supersteps for AR-05 and 201 supersteps for UK-05.

The Datalog query that evaluates the WCC algorithm is:

$$\begin{array}{l}\textbf{wcc}(x) \leftarrow \text{vertices}(x). \\ \textbf{wcc}(x, \text{MIN}(l)) \leftarrow \text{edges}(y,x), \text{wcc}(y,l).\end{array}$$

Query 14. WCC

The first rule initializes the component of a vertex to its id. The second rule is recursive and assigns to a vertex the component that is the minimum of its incoming neighbors. The evaluation stops when the component of all vertices don't change anymore.

PageRank is described in Section 4.2, and we copy the query here for ease of readability. We compare it against the customized implementation provided in the Giraph example package.

out-degree$(x, COUNT(y)) \leftarrow$ edges(x,y).
out-rank$(x, r/d) \leftarrow$ pagerank(x,r), out-degree(x,d).
pagerank$(x,1) \leftarrow$ superstep$(x,i), i == 0$.
pagerank$(x, SUM(r)) \leftarrow$ out-rank(y,r), superstep$(x,i), i < \sigma$.

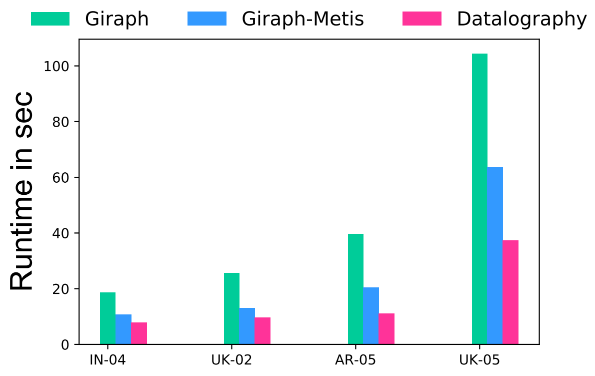
Query 15. PageRank

4.7.2 Runtime analysis

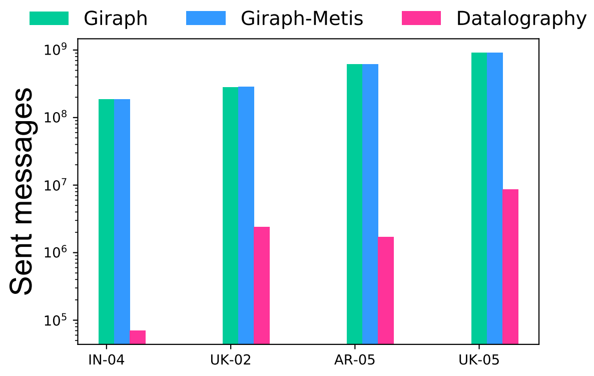
For all datasets and algorithms, we chose the number of super-vertices that offers the best performance for Datalography and Giraph-Metis. This decision does not affect the performance of Giraph as it is agnostic to the number of super-vertices. Giraph-Metis performs better than Giraph as less messages cross partition boundaries and are sent over the network. However, the number of messages and message bytes is exactly the same between them. Datalography achieves greater performance due to using super-vertices that reduce the number of messages sent and eager-aggregation that reduces the bytes of messages sent. In all experiments, we see larger gains for the largest dataset which is encouraging.

SSSP: Figure 4.14a shows the runtime performance of the three systems. We can see that Datalography outperforms Giraph for all datasets. Datalography offers greater speedup when compared to Giraph ranging from 2.4x to 3.6x, with the largest gains observed with AR-05 that has the smallest degree. When compared to Giraph-Metis the speedup is less and ranges from 1.4x to 1.9x with the greatest gains again for AR-05.

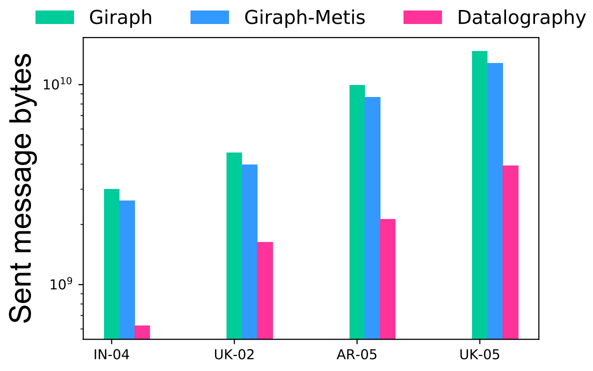
Figures 4.14b, 4.14c show how communication overheads affect runtime. Both Giraph and Giraph-Metis send the same number of messages and message bytes showing that employing smart partitioning of the input graph is beneficial for runtime. Datalography on the other hand, sends two orders of magnitude less messages highlighting the fact that combining partitioning with eager-aggregation can lead to greater benefits. The total size in bytes of sent messages does not observe such a steep difference compared to Giraph and



(a) Running time



(b) Aggregate number of messages



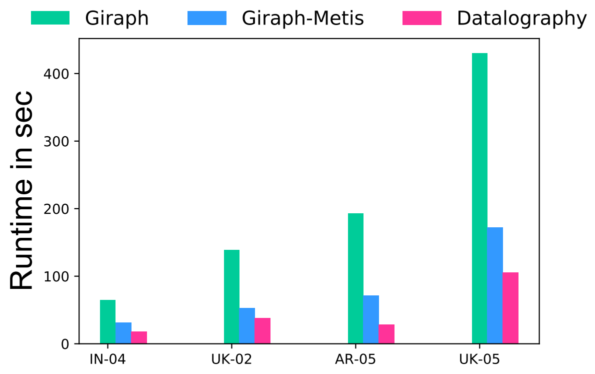
(c) Aggregate message bytes

Figure 4.14. SSSP: Performance comparison of Giraph, Giraph-Metis and Datalography

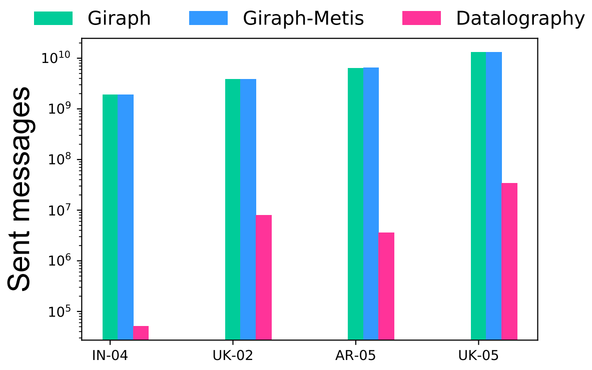
Giraph-Metis, $2.4x$-$4.2x$, as DATALOGRAPHY actually sends more information per message. Every message in DATALOGRAPHY contains the relation `sssp`, which has one tuple with 2 variables, compared two one double in Giraph and Giraph-Metis. Moreover, each message must include also the destination vertex inside the super-vertex the message is sent to since Giraph is agnostic of the internal vertices.

WCC: Again, DATALOGRAPHY outperforms Giraph and Giraph-Metis on all datasets (Figure 4.15a). The speedup observed is larger than with SSSP and ranges from $3.6x$ to $6.8x$ for AR-05 when compared to Giraph whereas when compared to Giraph-Metis the gains range from $1.4x$ to $2.5x$. The climb in the plot observed at UK-02 is due to the large number of supersteps WCC required. Figures 4.15b, 4.15c show that Giraph and Giraph-Metis send 2-3 orders of magnitude more messages than DATALOGRAPHY and $7.5-13.6$ times more message bytes. The messages are again larger than the ones sent with the Giraph implementations and consist of a tuple of the `wcc` with 2 variables and the information of the internal vertex.

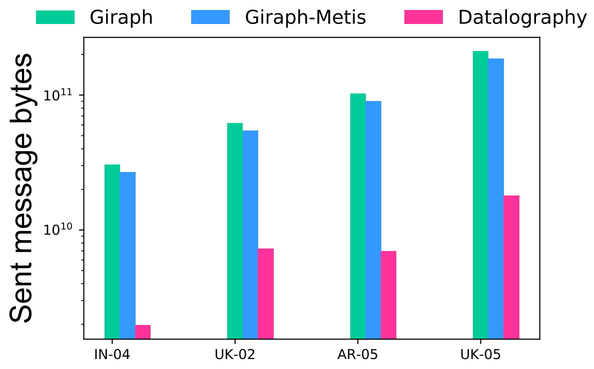
PageRank: PageRank is different from the other algorithms in that the amount of data being processed during every iteration does not shrink as the algorithm nears termination or converges, as every vertex at every superstep computes a new rank and sends messages to its outgoing neighbors. We see in Figure 4.16a that DATALOGRAPHY outperforms Giraph by a factor of $2.3-3.3$ times whereas it outperforms Giraph-Metis by a factor of $1.1-2.3$ times. Figures 4.16b and 4.16c show the communication overhead comparison. If we focus on UK-02, the maximum expected number of messages is $edges \times supersteps$ which is 5×10^9. Both Giraph and Giraph-Metis send as many messages as seen in Figure 4.16b. DATALOGRAPHY on the other hand, sends 6×10^6 messages. The maximum expected number of bytes is $edges \times iterations \times 16bytes$ which is 95×10^9. Again, both Giraph and Giraph-Metis send this many bytes whereas DATALOGRAPHY sends 13×10^9 bytes. The size of the messages is larger for DATALOGRAPHY as a message consists of the relation `msg` that has two variables. Moreover, each message must include the



(a) Running time

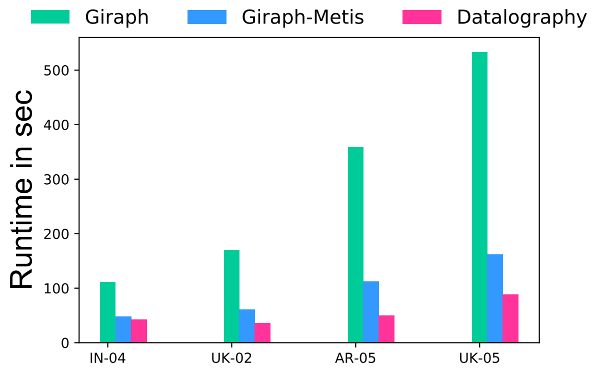


(b) Aggregate number of messages

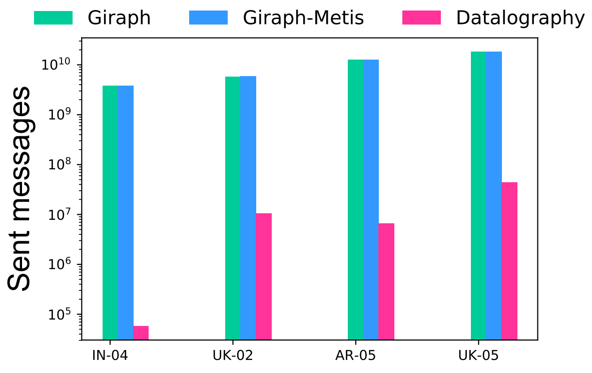


(c) Aggregate message bytes

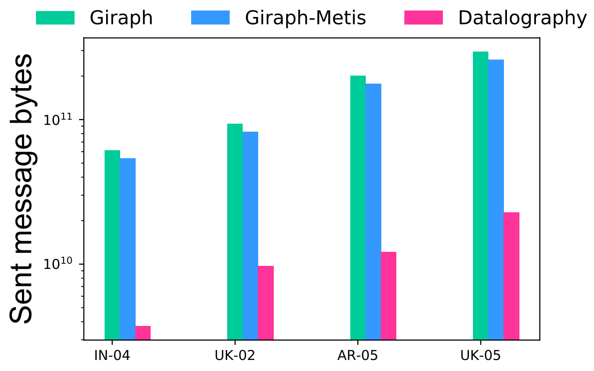
Figure 4.15. WCC: Performance comparison of Giraph, Giraph-Metis and DATALOGRAPHY



(a) Running time



(b) Aggregate number of messages



(c) Aggregate message bytes

Figure 4.16. PageRank: Performance comparison of Giraph, Giraph-Metis and DATALOGRAPHY

information of the destination vertex in the super-vertex.

4.7.3 Scaling super-vertices

In these experiments we scale the number of super-vertices to show the effect they have on runtime and communication. We observe a tradeoff between parallelism and communication overheads as too few super-vertices do not take advantage of parallelism whereas too many super-vertices hinder the computation and communication optimizations otherwise offered. For each dataset, we chose the number of super-vertices such that it would be a multiple of 27 which is the maximum number of Giraph workers our hardware specifications can support. The lower limit is imposed by Hadoop's file loader limit on

Table 4.3. Number of super-vertices created for every dataset.

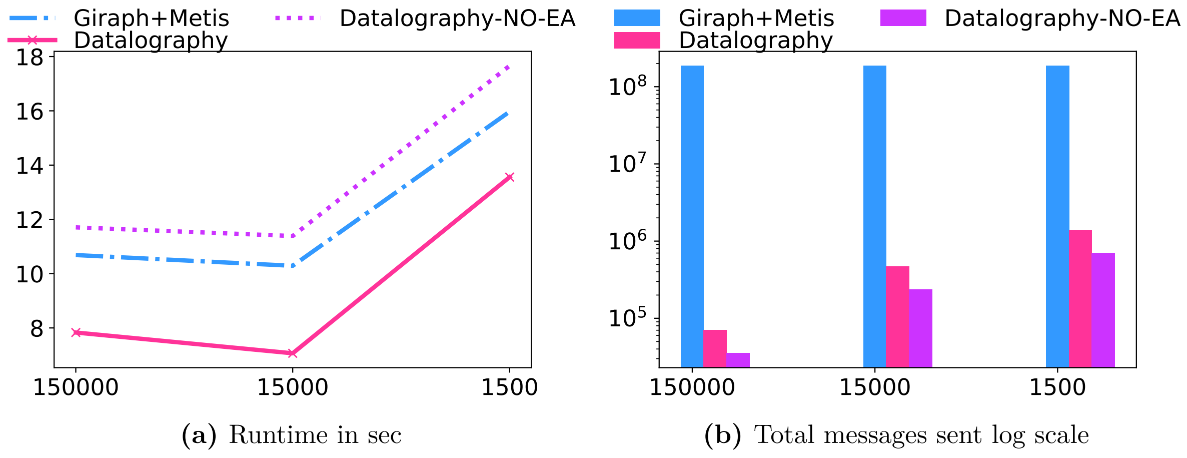
Dataset	**1500**	**15000**	**150000**
IN-04	54	486	4941
UK-02	135	1215	12339
AR-05	216	1620	13608
UK-05	324	2916	23328

the size of a line: DATALOGRAPHY input format has a super-vertex per line with all its information such as internal vertices and edges. The upper limit is imposed by ParMetis and the number of partitions it can create. Given these restrictions, the number of vertices included in each super-vertex ranges roughly 1500 - 15000. Table 4.3 shows the number of super-vertices created for each dataset.

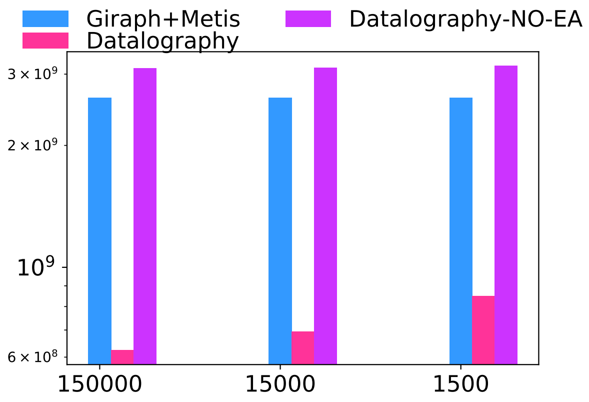
For every algorithm and dataset, we compare the running time and communication overhead of Giraph-Metis, DATALOGRAPHY and DATALOGRAPHY-NO-EA that is not using the optimization of eager-aggregation. We expect the latter to have similar or worse performance than Giraph-Metis as eager-aggregation offsets the overhead of DATALOGRAPHY sending larger messages in bytes when compared to Giraph. Eager-aggregation or no should not affect the number of messages but reduce the size of them as its effect is that multiple neighbors sending messages to the same vertex, aggregate their messages and instead send only one. In both cases, since Giraph is agnostic of the internal vertices, it sees the same number of vertices.

When memory is an issue, such as with UK-05 that is our largest dataset, a smaller number of super-vertices offers better performance.

Figures 4.17, 4.18, 4.19, 4.20 show the effect of super-vertices and eager-aggregation on SSSP for every dataset. With the exception of IN-04, DATALOGRAPHY-NO-EA performs similar or better than Giraph-Metis. All three systems observe the same trend with the best running time being at 15000 vertices per super-vertex. Only for UK-05 the running time is better for the smallest number of super-vertices. The number of messages sent is the same between DATALOGRAPHY and DATALOGRAPHY-NO-EA as expected, since eager-aggregation affects only the size of messages and they are both less than Giraph-Metis. Moreover, the number increases as the number of super-vertices increases. The message bytes sent however are more for DATALOGRAPHY-NO-EA than both DATALOGRAPHY and Giraph-Metis and also increase with the number of super-vertices.



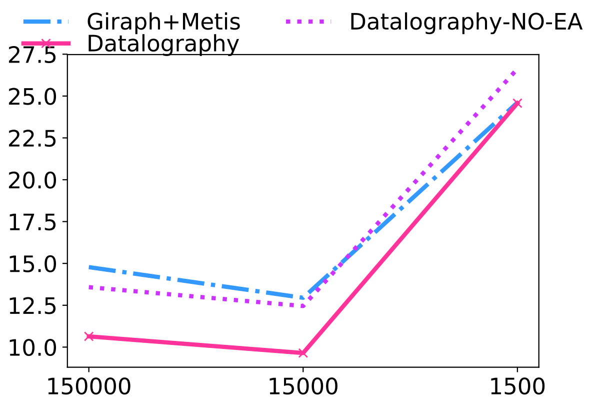
(a) Runtime in sec **(b)** Total messages sent log scale



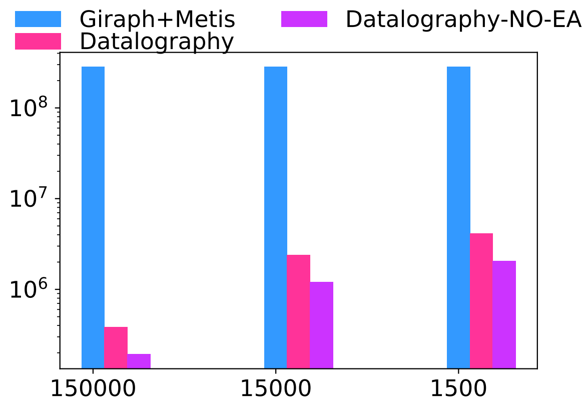
(c) Total message bytes log scale

Figure 4.17. SSSP IN-04: Runtime and communication per super-vertex size.

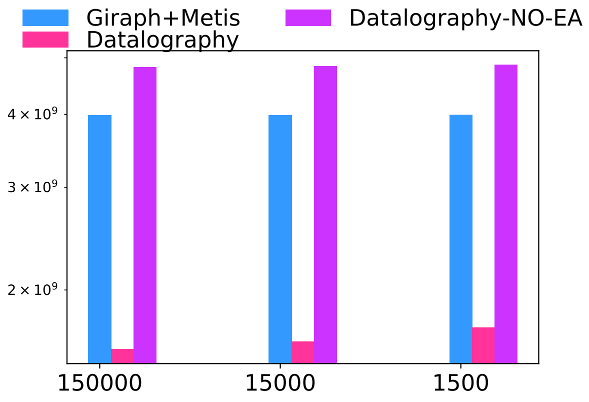
Figures 4.21, 4.22, 4.23, 4.24 show the effect of super-vertices and eager-aggregation on WCC for every dataset. DATALOGRAPHY-NO-EA performs worse than both Giraph-Metis and DATALOGRAPHY for all super-vertex sizes. This highlights the necessity of eager-aggregation and that using smart partitioning by itself is not enough. All three systems observe the same trend with respect to the scaling of super-vertices with the best



(a) Runtime in sec



(b) Total messages sent in log scale

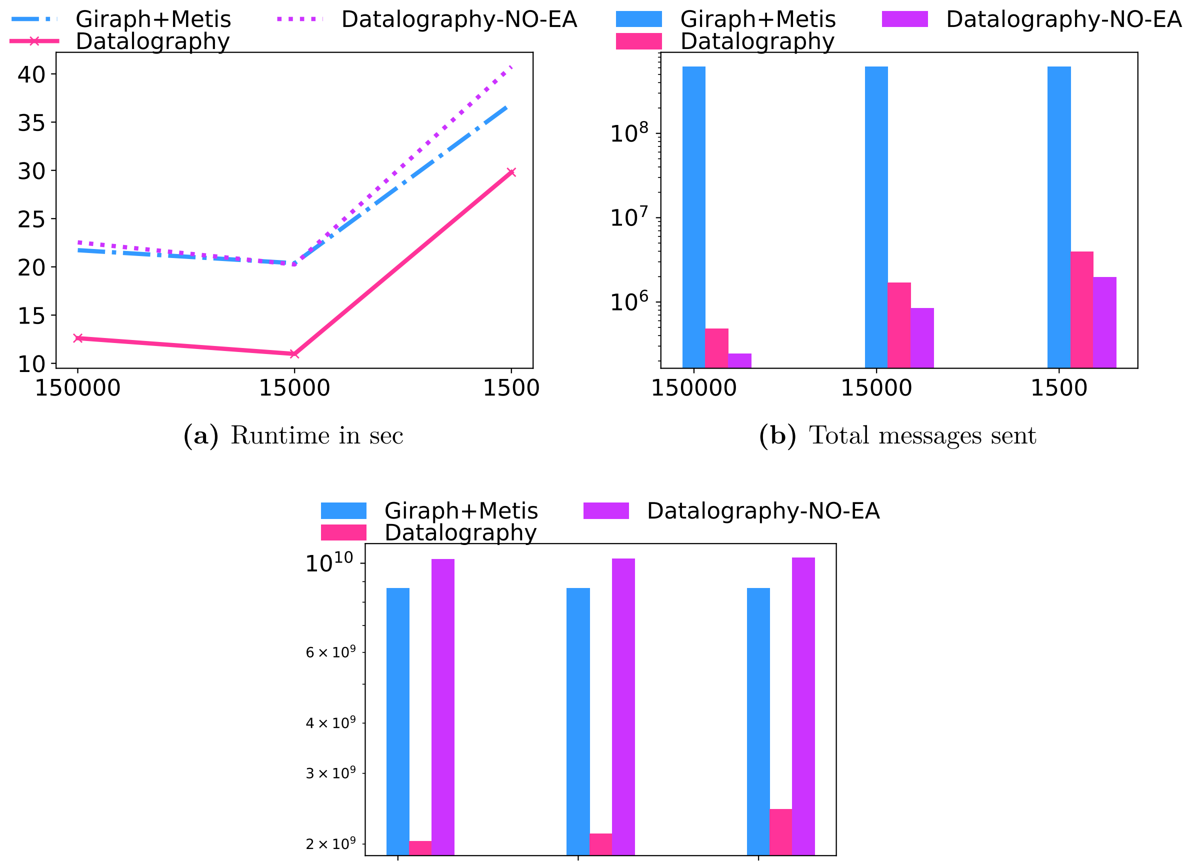


(c) Total message bytes in log scale

Figure 4.18. SSSP UK-02: Runtime and communication per super-vertex size.

running time being at 15000 vertices per super-vertex. Like with SSSP, the number of messages sent is the same between DATALOGRAPHY and DATALOGRAPHY-NO-EA and they are both less than Giraph-Metis whereas the message bytes are far less for DATALOGRAPHY than both DATALOGRAPHY-NO-EA and Giraph-Metis.

With PageRank, both runtime and communication follow the same trend as before. In Figures 4.25, 4.26, 4.27, 4.28 we see that the best time is observed at 15000 vertices per super-vertex and DATALOGRAPHY-NO-EA performs worse than DATALOGRAPHY and Giraph-Metis. Communication overheads increase with the size of super-vertices and DATALOGRAPHY sends the smallest number and bytes of messages.



(a) Runtime in sec

(b) Total messages sent

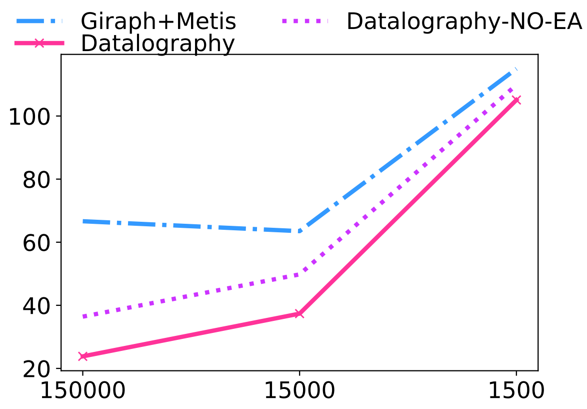
(c) Total message bytes

Figure 4.19. SSSP AR-05: Runtime and communication per super-vertex size.

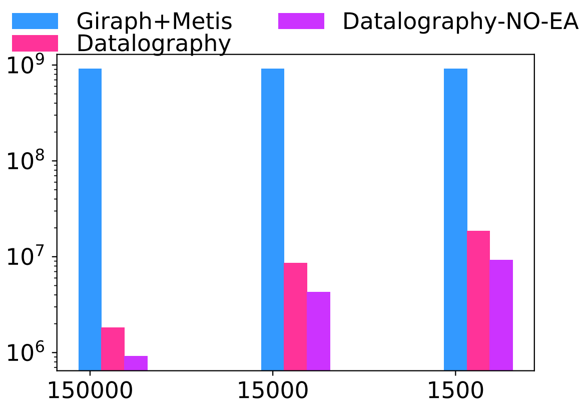
4.8 Related work

Datalog has recently witnessed a resurgence [37] appearing as the language of choice in commercial databases (LogicBlox [9]) as well as research projects choosing declarative semantics in areas such as program analysis [12] and security [49] to name a few. Below, we focus on related works that use Datalog in the context of distributed and parallel computation.

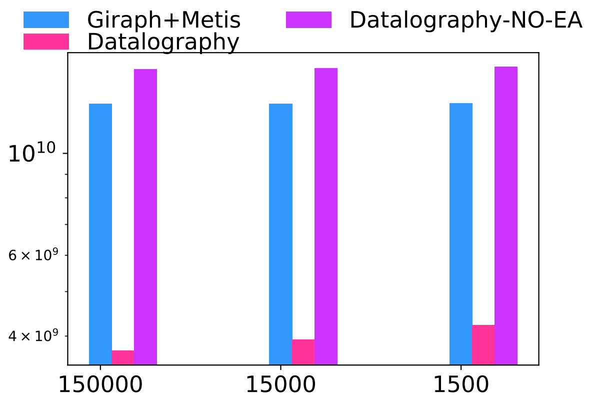
Distributed Datalog languages. Bloom [6] and Bloom^L [20] suggest the use of Datalog as a distributed programming language whose monotonicity semantics allow for



(a) Runtime in sec



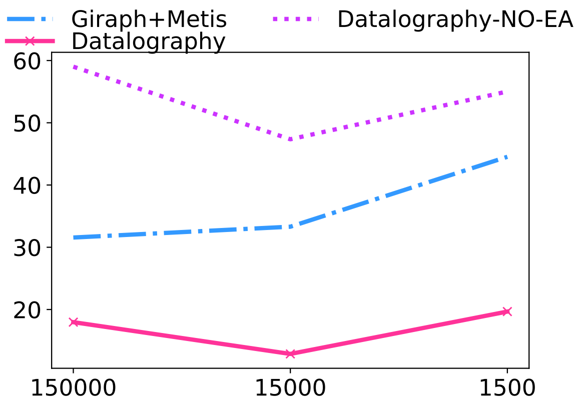
(b) Total messages sent in log scale



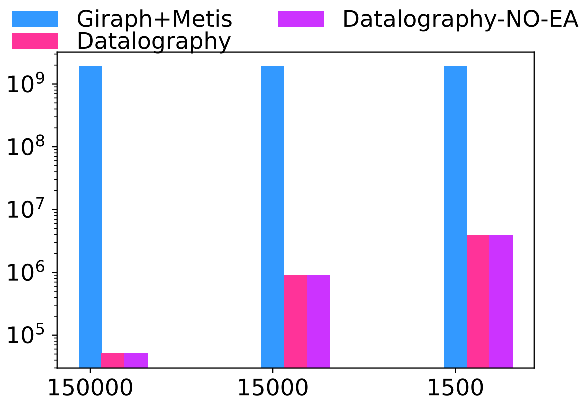
(c) Total message bytes in log scale

Figure 4.20. SSSP UK-05: Runtime and communication per super-vertex size.

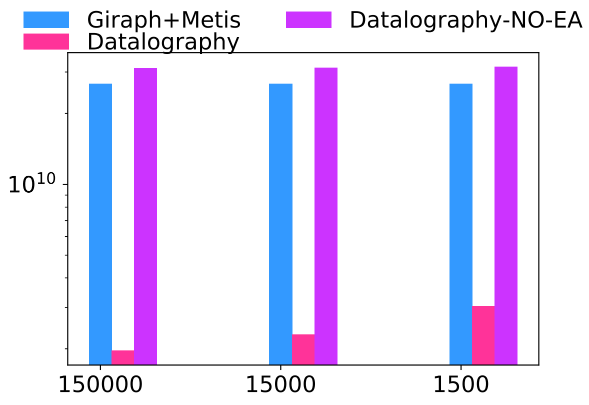
coordination-free evaluation. NDlog [45] for declarative networking shares similarities with our use of location specifiers in rules. In their context, location specifiers represent network addresses. They define locality of rules in a similar manner to ours to allow evaluation on networks that are not a full mesh. They support monotonic aggregates defined on partial ordered domains and stratified aggregation for SUM as in Query 2. Negation is not supported. BOOM analytics [5] implemented the Hadoop MapReduce framework and HDFS storage system using Overlog. Dyna [22] is a language extension of Datalog to support modern AI analytics. Yedalog [16], based on Dyna, suggests Datalog as a signle language of choice for expressing both computation on semi-structured data and dataflow



(a) Runtime



(b) Total messages sent in log scale



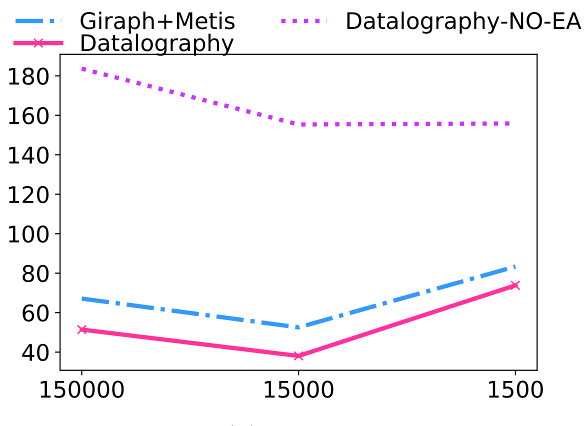
(c) Total message bytes in log scale

Figure 4.21. WCC IN-04: Runtime and communication per super-vertex size.

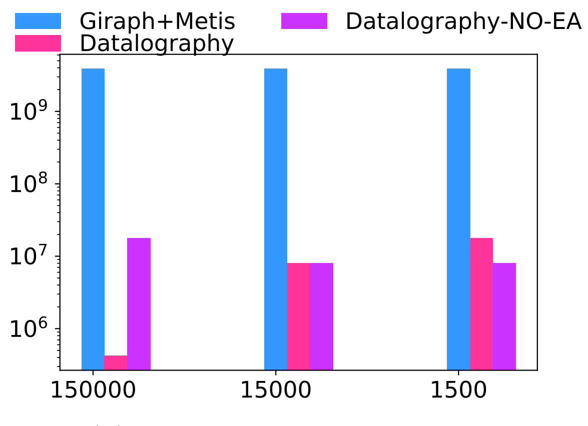
on batch-processing frameworks. [75, 4] looked into reducing the iterations of recursion in distributed transitive closure through non-linear recursion by effectively reducing the data volume exchanged between workers. It is not clear how to apply this idea to general Datalog programs.

Declarative large-scale graph analytics

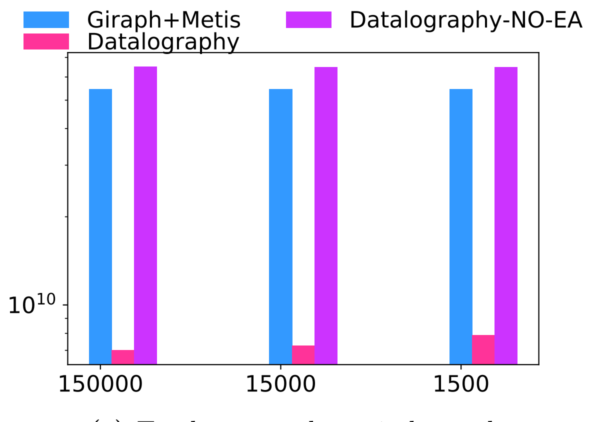
Large-scale Datalog frameworks tend to use bespoke engines or modify existing system to add support for iterative graph computation. Interestingly, other work [13] explored the opposite direction, transforming existing imperative programs expressed with the *Vertex-Centric* paradigm into Datalog analytics that then execute an a relational



(a) Runtime



(b) Total messages sent in log scale

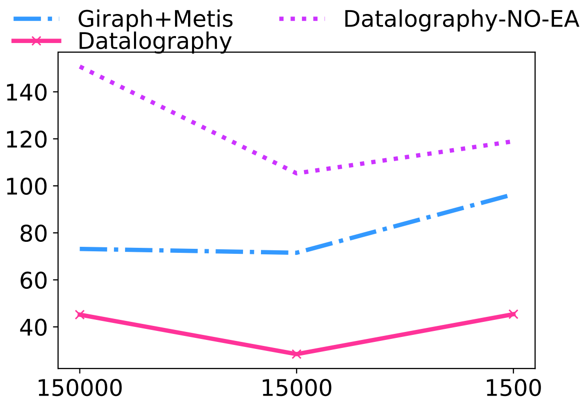


(c) Total message bytes in log scale

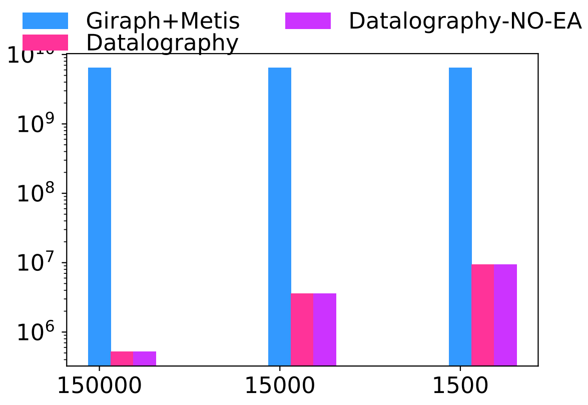
Figure 4.22. WCC UK-02: Runtime and communication per super-vertex size.

dataflow system [11].

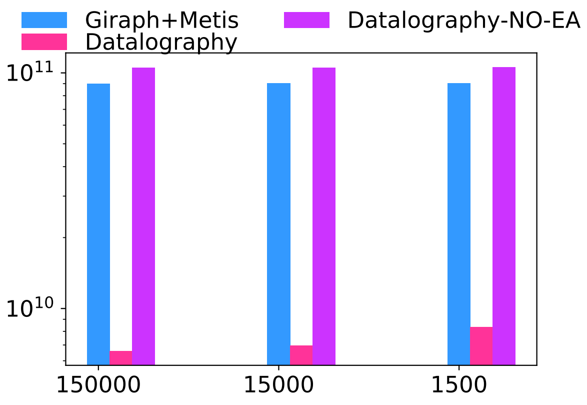
Socialite [62] and Myria [71] are specialized engines catered to the distributed evaluation of Datalog queries. In Socialite, the input graph is sharded among workers based on a partition key provided by the user. The workers evaluate rules asynchronously and communicate via messages until no new facts can be deduced. Myria utilizes a distributed relational database engine that consists of one master and multiple workers, and hence, query plans comprise relational algebra operators that are partitioned across workers. It supports both synchronous and asynchronous execution. Both approaches support in a clean way a subset of recursive aggregates that are associative, commutative



(a) Runtime



(b) Total messages sent in log scale

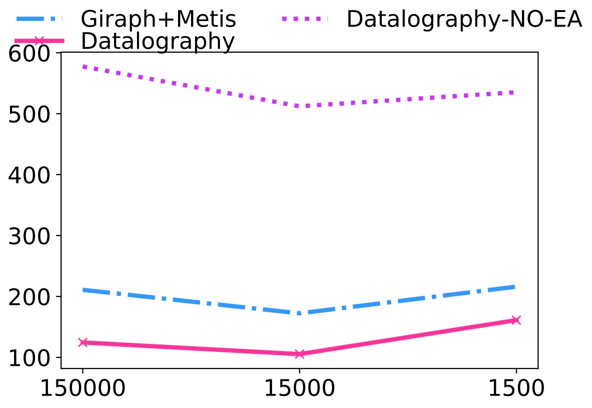


(c) Total message bytes in log scale

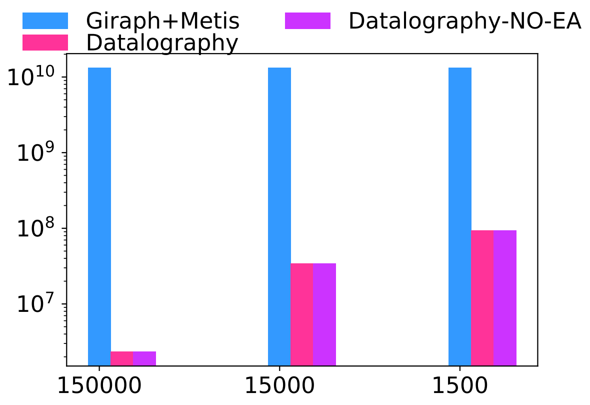
Figure 4.23. WCC AR-05: Runtime and communication per super-vertex size.

and idempotent (e.g., MIN,MAX). Socialite supports non-meet aggregate operations such as SUM by embedding a Datalog query inside an imperative loop, effectively running n Datalog programs. Myria supports monotonic SUM,COUNT aggregates defined using partial orders and bag-semantics. In contrast, DATALOGRAPHY supports both monotonic and non-monotonic aggregates under set-containment, with formal semantics, without the use of imperative code.

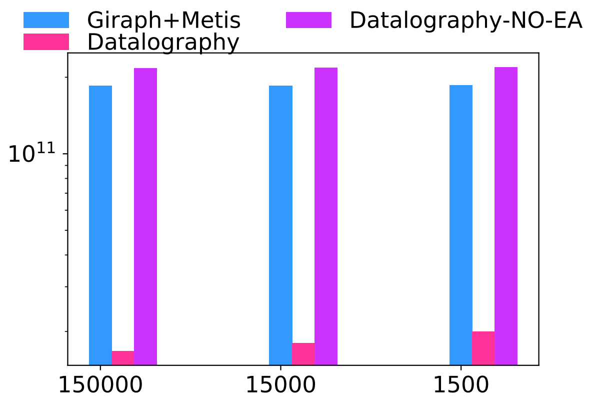
BigDatalog [65] is built on top of Spark [76], a general platform for large-scale analytics. Spark cannot support recursion out of the box and BigDatalog had to implement optimizations on the Spark runtime such as a specialized RDD implementation and a



(a) Runtime



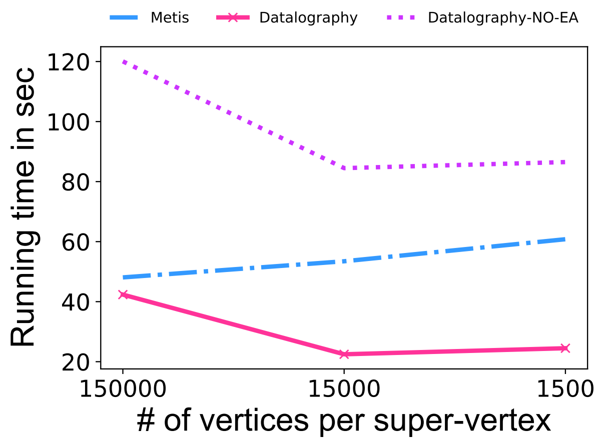
(b) Total messages sent in log scale



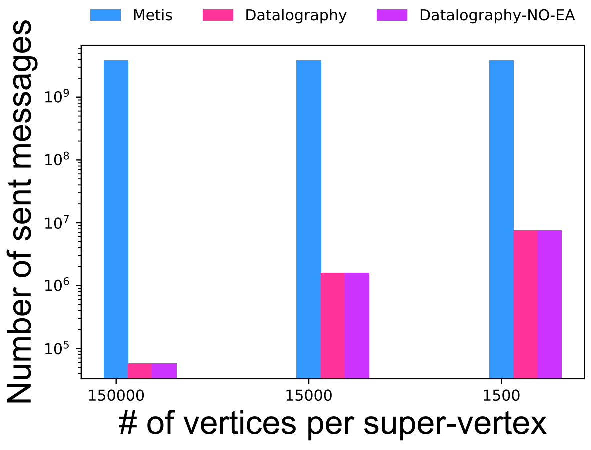
(c) Total message bytes in log scale

Figure 4.24. WCC UK-05: Runtime and communication per super-vertex size.

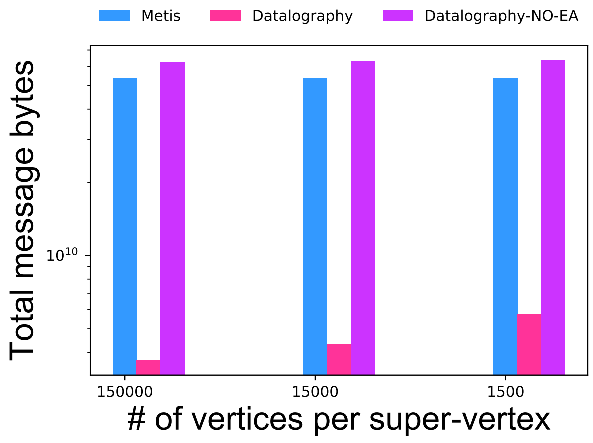
scheduler aware of iterations. DATALOGRAPHY in contrast, is built on top of Giraph, a large-scale graph processing system where iterative computation is inherently supported. BigDatalog implements the approach of DeALS [66] for recursive aggregates by combining monotonic recursive versions, i.e., *mmin*, *mmax*, *mcount*, *msum*, with a final non-recursive aggregate. We follow their semantics for monotonic aggregates, but additionally support non-monotonic aggregates such as the one used in PageRank.



(a) Runtime



(b) Total messages sent in log scale

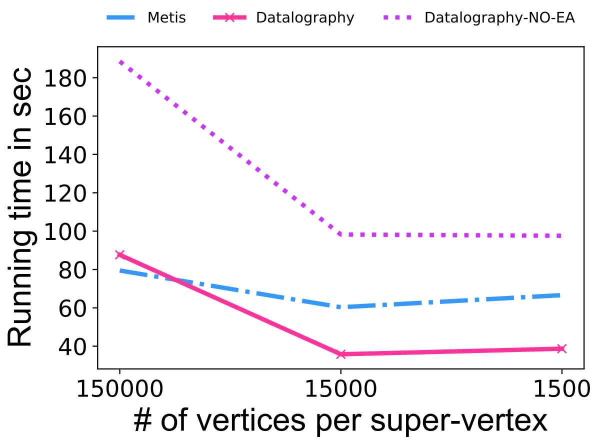


(c) Total message bytes in log scale

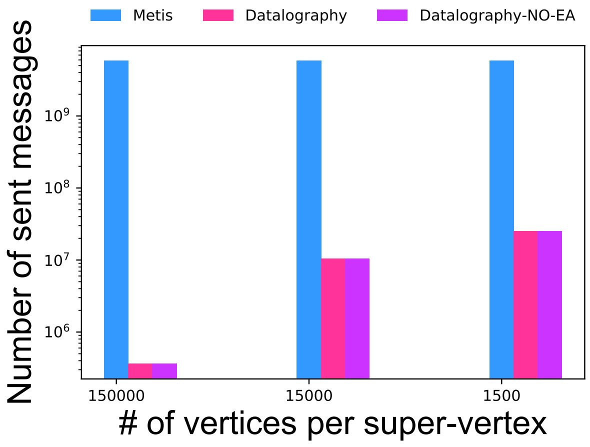
Figure 4.25. PageRank IN-04: scaling super-vertices

4.9 Chapter summary

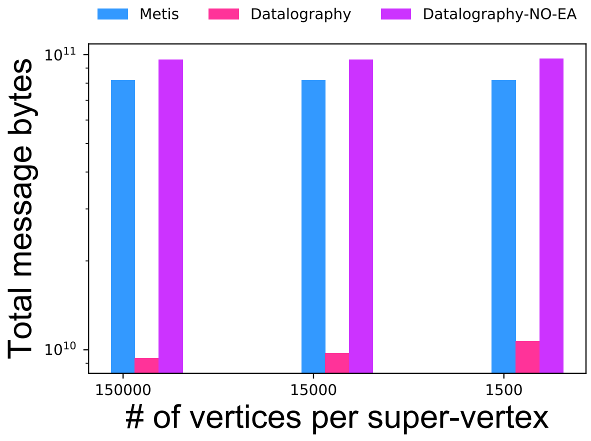
In this chapter, we presented DATALOGRAPHY, the first Datalog compiler and evaluation engine on *Vertex-Centric* frameworks. Developers can take advantage of the concise,declarative nature of Datalog and automatically optimized recursive evaluation. Moreover, our approach does not require Datalog queries to be expressed according to the *Vertex-Centric* paradigm, any general Datalog query is supported. To this extend, we designed a compiler that, through logical query rewritings, transforms an initial query



(a) UK-02



(b) Total messages sent in log scale

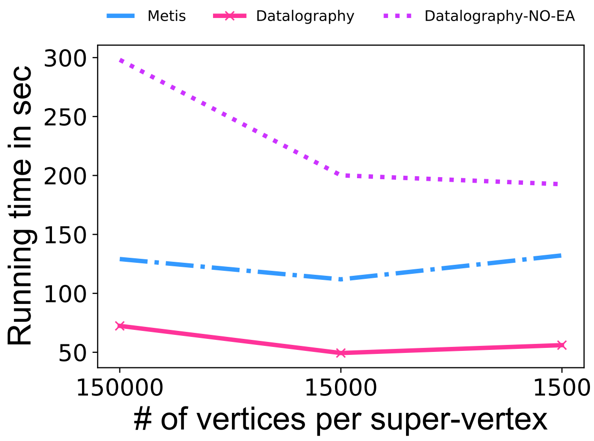


(c) Total message bytes in log scale

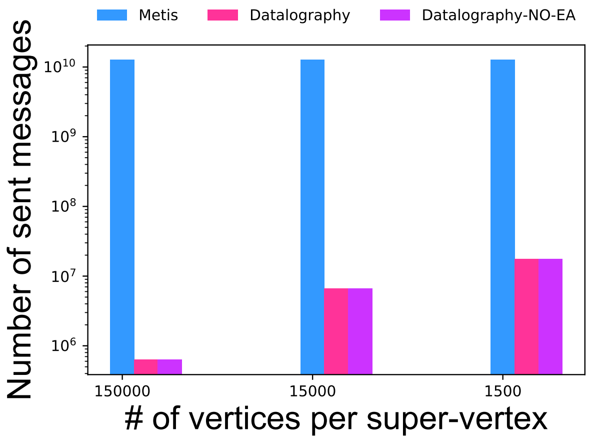
Figure 4.26. PageRank UK-02: scaling super-vertices

into an equivalent one, amenable to distributed evaluation on *Vertex-Centric* engines. We propose the abstraction of super-vertices, that open the way to novel optimizations such as source-side combiners. We implemented DATALOGRAPHY on Apache Giraph and our experimental results confirm that DATALOGRAPHY outperform imperative Giraph graph analytics.

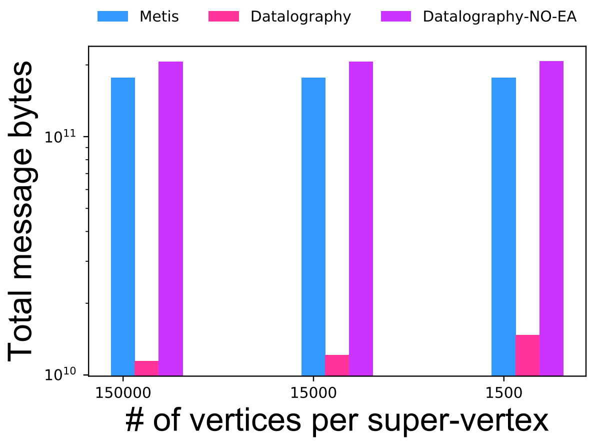
This chapter contains material from ”Datalography: Scaling datalog graph analytics on graph processing systems” by Walaa Eldin Moustafa, Vicky Papavasileiou, Ken Yocum, Alin Deutsch which appears in Proceedings of International Conference on Big Data, pages



(a) Runtime



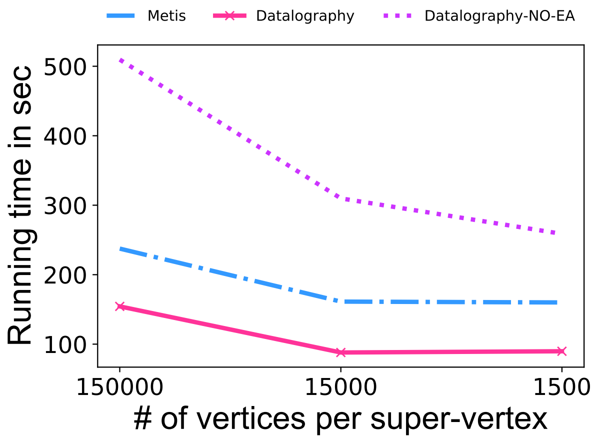
(b) Total messages sent in log scale



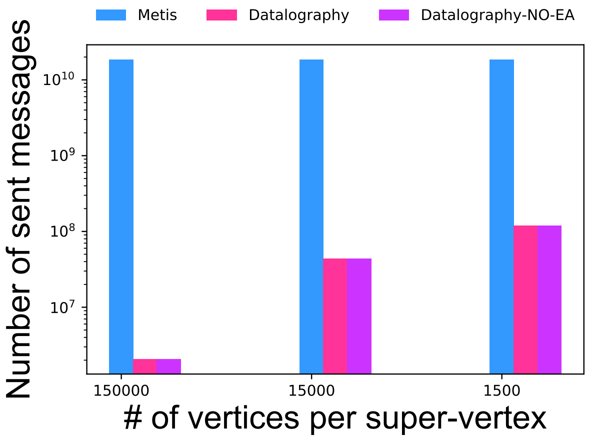
(c) Total message bytes in log scale

Figure 4.27. PageRank AR-05: scaling super-vertices

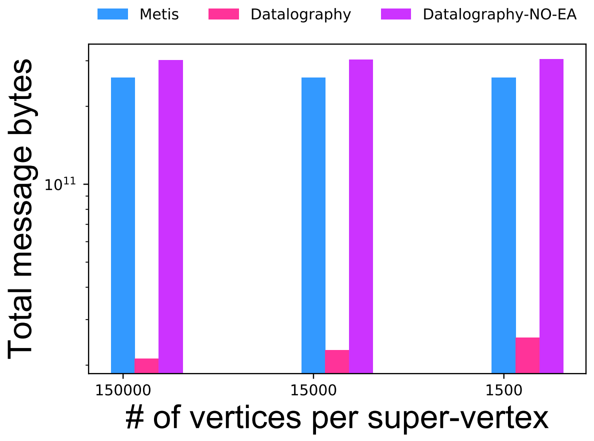
56 –65, 2016. The dissertation author was a primary investigator and co-author of this paper.



(a) UK-05



(b) Total messages sent in log scale



(c) Total message bytes in log scale

Figure 4.28. PageRank UK-05: scaling super-vertices

Chapter 5

Ariadne: Online Provenance Querying For Big Graph Analytics

5.1 Introduction

Large-scale graph processing engines like Giraph [1], GraphX [31], GraphLab [46] and Pregel [48] are popular for analyzing data generated from systems in biology, finance, and social networks. They offer high-level programming abstractions that make it easy to author scalable graph analytics. The analytics evaluate on an input graph for a given number of iterations or until a fixed point. Graph analytics range from well-known graph computations like shortest paths and connected components to ML algorithms like clustering and recommenders. These engines are *Vertex-Centric* (VC) as they follow the *"Think-like-a-vertex"* programming model where a single program repeatedly runs on each vertex.

Developers spend much of their time cleaning and exploring data. These activities often involve the interplay between analytics and the large-scale data on which they run. For instance, a data worker may look for aberrant algorithm behavior during code development as new data sets arrive. This can involve "crash culprit determination" – finding input data elements that caused code to fail. However, there are other equally important non-crash related activities. These include asserting invariants in the behavior of the analytic and checking for data formats / ranges. In addition, the emergence of approximate versions of graph analytics [54, 63] allows scientists to trade accuracy for runtime. However it remains difficult to reason about how improvements in runtime affect losses in accuracy [63].

Provenance can enable such exploration through tools that allow developers to easily search and query the way data changes during an analytic. Indeed, prior work illustrates the power of pinpointing data inputs responsible for a system crash or exception in data-intensive processing. While some of these facilities are available today for systems that run data-parallel workflows (Hadoop, Spark, etc.) – "guard predicates" [33] and provenance tracing[38, 44, 55] – none address them for graph processing systems.

Though vertex-centric graph programs are relatively simple, they present challenges to effectively capturing and querying provenance. Vertex-centric graph processing systems [48, 1, 46] repeatedly execute the same code on each graph vertex for tens to hundreds of iterations. Our experiments show that the provenance of graph analytics can be $10x$ larger than the input graph whereas the provenance of Spark [76] analytics amounts to $30\% - 50\%$ of the input [38]. Moreover, a tracing query can yield results that include a majority of the input graph. Many graph analytics diffuse information as they run, sending and receiving messages from nearby vertices, e.g., PageRank sends weight to neighbors, shortest path tracks path length. Thus provenance traces grow quickly; the answer to a backwards trace could be the entire input graph.

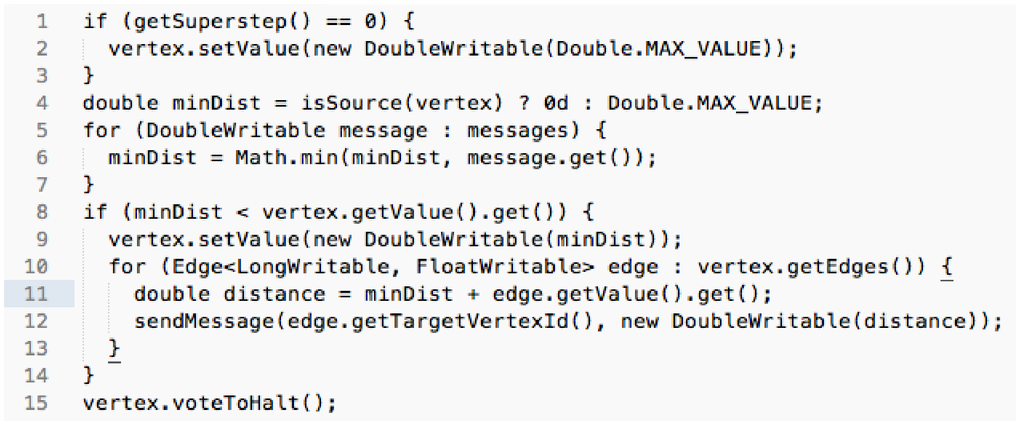
This paper presents Ariadne, a system to capture, store, and analyze provenance for vertex-centric graph processing engines. Ariadne defines a provenance querying language (PQL) that developers use to declaratively customize provenance capturing and analysis. Ariadne enables an important class of queries that allow *online* provenance analysis that executes simultaneously with an analytic. At the end of the computation, the results of the analytic and the provenance queries (capturing and analysis) exist. Moreover, because Ariadne represents the captured provenance as a graph, it is immediately available for further, repeat analysis on the same engine.

This work provides the following contributions:

- A formal provenance model for vertex-centric graph computations and a simple, efficient physical representation of the resulting provenance graph.
- A declarative, concise datalog-based language, Provenance Query Language (PQL). Additionally, this work presents the theoretical underpinnings of PQL, describing a PQL query taxonomy that includes query subclasses that admit efficient querying strategies that operate on small subsets of provenance.
- A provenance query taxonomy that identifies a query subclass ('forward direct', Section 5.4.2) that allows novel *online* querying and *tailored* capture. We illustrate the generality of this subclass with a suite of queries to analyze algorithm behavior and perform invariant/audit checking.
- A system architecture that allows queries of this class to execute as ordinary vertex programs without modification to the graph processing engine itself.
- A Giraph-based prototype with which we measure query performance across a range of analytics and real-world graphs. Online query and capture (running multiple PQL datalog queries during an analytic) incurs a $2x$ over the baseline graph analytic. When ARIADNE evaluates only queries (no capture), the average overhead is $1.3x$ across our analytics and datasets.

5.2 Running example and System overview

This section overviews ARIADNE's functionality using a motivating example. Assume Single-Source Shortest Paths (SSSP) is a common computation for a developer that she would like to tune for performance. Let's assume our developer (Alice) doesn't care about the exact distance of vertices from a source, an approximation with a small error is



```
1  if (getSuperstep() == 0) {
2    vertex.setValue(new DoubleWritable(Double.MAX_VALUE));
3  }
4  double minDist = isSource(vertex) ? 0d : Double.MAX_VALUE;
5  for (DoubleWritable message : messages) {
6    minDist = Math.min(minDist, message.get());
7  }
8  if (minDist < vertex.getValue().get()) {
9    vertex.setValue(new DoubleWritable(minDist));
10   for (Edge<LongWritable, FloatWritable> edge : vertex.getEdges()) {
11     double distance = minDist + edge.getValue().get();
12     sendMessage(edge.getTargetVertexId(), new DoubleWritable(distance));
13   }
14 }
15 vertex.voteToHalt();
```

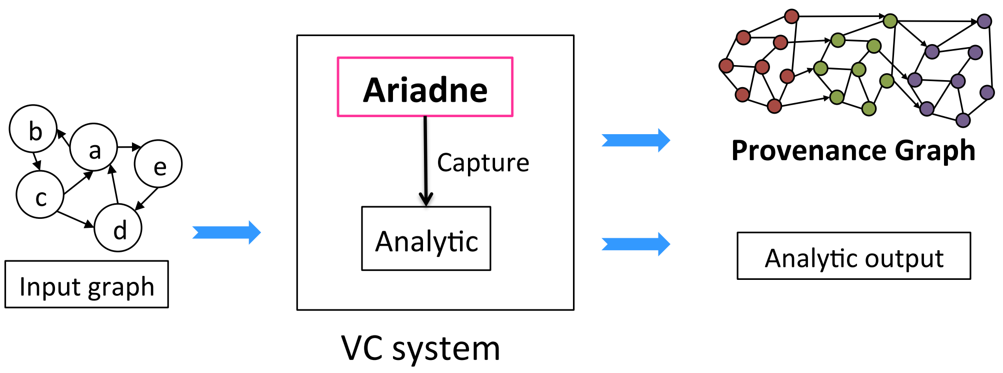
Figure 5.1. Running example: Giraph SSSP

fine. One way to create an approximate version of a graph analytic is to only message neighbors on large updates. This increases the likelihood that some vertices receive no messages and stop computing earlier. How could she use ARIADNE to analyze the behavior of an analytic to determine the applicability of such an optimization?

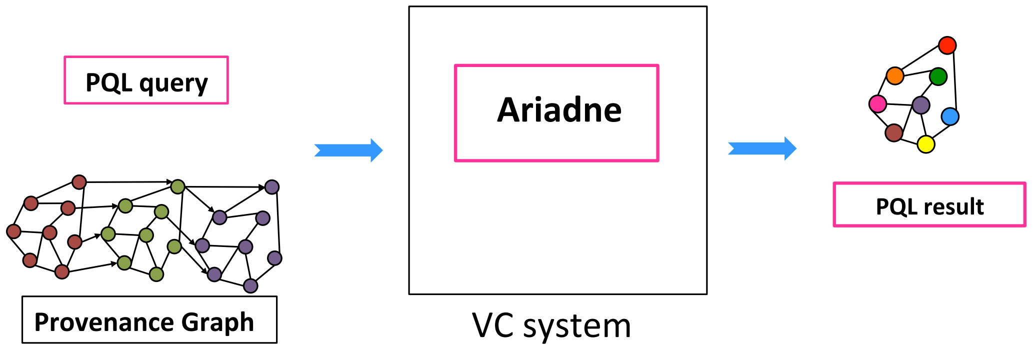
Figure 5.1 shows the SSSP program in Giraph's example library. In superstep 0 every vertex initializes its value (distance to source) with MAX.DOUBLE. Every subsequent superstep involves the following steps: If the current executing vertex is the source, the distance to itself is 0, else it is MAX.DOUBLE. Line 6 selects the minimum value between the current distance and the one received from neighbors by iterating over the messages received. If the new distance is smaller than the current distance, Line 9 updates the value of the current vertex and Line 12 sends a message to the neighbors with the new distance plus the respective edge weight.

To analyze the behavior of SSSP at scale, Alice needs a provenance management tool to look at the vertex values and messages at every superstep. This way, Alice can investigate how much the distance of each vertex changes over the course of computation. Moreover, she can determine the number of vertices whose distance doesn't change and the number of consecutive supersteps for which this phenomenon occurs.

Traditionally, provenance management at scale follows a pipeline that involves



(a) Declarative provenance capturing.



(b) Declarative offline provenance querying.

Figure 5.2. Architecture of ARIADNE that improves traditional offline provenance usage.

provenance capturing, where no user customization is possible, storing of the captured provenance, and offline querying it using imperative, tracing functionality. ARIADNE improves upon this to allow developers to customize provenance capturing and offline analysis in a declarative manner. Figure 5.2a shows how Alice can issue a declarative query alongside her *unchanged* graph analytic to capture its provenance. At the end of computation, both the graph analytic's result as well as the custom provenance information exist. Then, Figure 5.2b shows how Alice can use ARIADNE to analyze the captured provenance, again via a declarative query on the same *Vertex-Centric* system the graph analytic ran.

Besides improving the traditional approach of provenance capturing and offline analysis, ARIADNE enables a novel methodology where provenance is analyzed in an

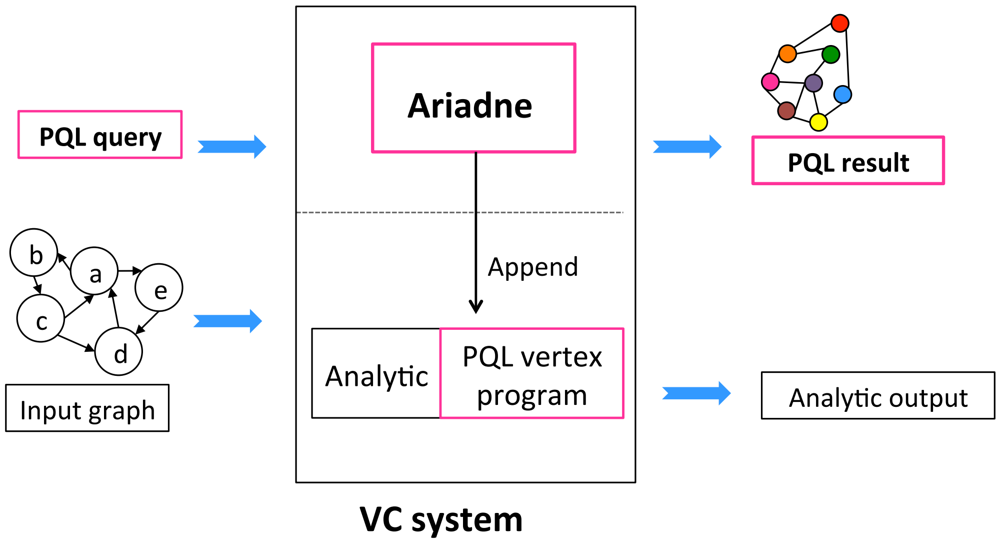
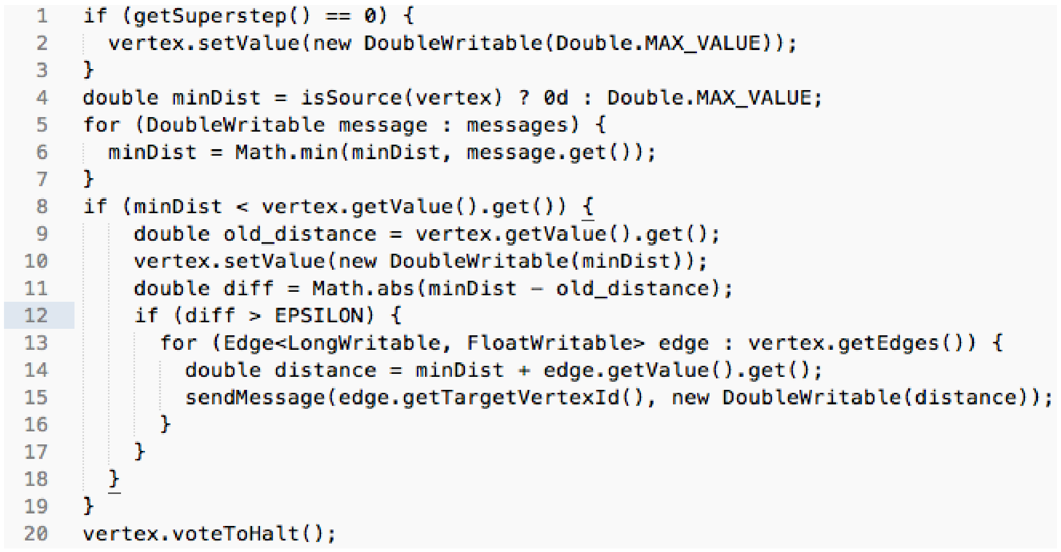


Figure 5.3. Online provenance querying using Ariadne.

online manner, while the analytic is executing. To achieve this, Ariadne appends the vertex program with provenance query evaluation, as shown in Figure 5.3, so that at every superstep a vertex evaluates the graph analytic as well as the provenance query. Note, the original graph analytic is unchanged from the perspective of the developer. Then, at the end of computation, both the graph analytic result and the provenance query result exist.

Online provenance querying offers a shortcut to provenance management when a developer knows upfront what information she is interested in. Alice can simply capture information of the analytic execution such as vertex values, or she can analyze the information to find, for example, the number of consecutive supersteps a vertex value doesn't change. In our scenario, Alice runs SSSP alongside Query 16 (described in Section 5.4) and determines she can she can use a threshold, ϵ, as a convergence criterion in an optimized version of SSSP, where a vertex sends its new distance to its neighbors, only if the change is larger than ϵ. If the change is less than ϵ, the vertex is considered to have converged. The optimized SSSP is in Figure 5.4.



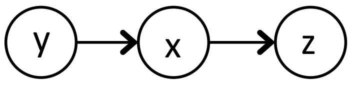
```
if (getSuperstep() == 0) {
  vertex.setValue(new DoubleWritable(Double.MAX_VALUE));
}
double minDist = isSource(vertex) ? 0d : Double.MAX_VALUE;
for (DoubleWritable message : messages) {
  minDist = Math.min(minDist, message.get());
}
if (minDist < vertex.getValue().get()) {
    double old_distance = vertex.getValue().get();
    vertex.setValue(new DoubleWritable(minDist));
    double diff = Math.abs(minDist - old_distance);
    if (diff > EPSILON) {
      for (Edge<LongWritable, FloatWritable> edge : vertex.getEdges()) {
        double distance = minDist + edge.getValue().get();
        sendMessage(edge.getTargetVertexId(), new DoubleWritable(distance));
      }
    }
  }
}
vertex.voteToHalt();
```

Figure 5.4. Running example: Giraph SSSP with convergence criterion

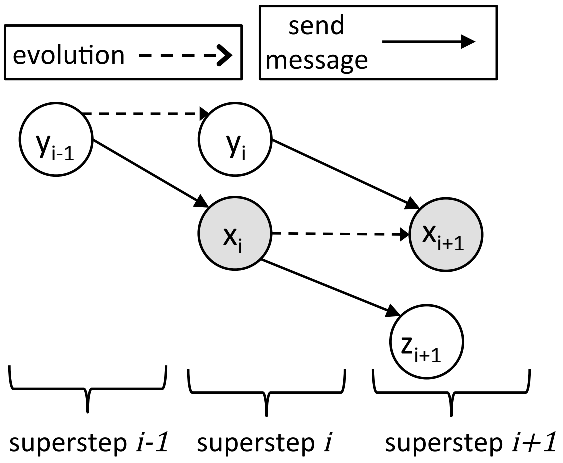
5.3 Provenance model

In this section we present the provenance data model of Ariadne that is the same as the data model of the analytics for which provenance is captured. This has the benefit that provenance can be queried on the same engine the analytic ran and using the same language (more on this in the next section).

Ariadne models provenance information as a graph, where a node represents a vertex computation at a specific superstep. The provenance information on every node can capture its value at every superstep, the messages it sent and received, and the value of its edges. Moreover, it can be derivations of these in the form of provenance query results. For example, Alice is not interested in the actual distances a vertex has in every superstep but rather in how much the distance changed. In addition to nodes, the provenance graph has two kinds of edges: The first, the *send/receive message* edges, connects nodes that represent neighboring vertices in the input graph and shows the message exchange between them. The second, the *evolution* edges, connects nodes that represent the same vertex but at different supersteps providing information about when a vertex was active during computation. Edges of the input graph are represented as nodes in the provenance graph.



(a) Input graph



(b) Provenance graph

Figure 5.5. Provenance graph for SSSP

As an example, consider the excerpt of an input graph in Figure 5.5a with vertex x, its incoming neighbor y and outgoing neighbor z. Alice executes SSSP on this graph and captures in the provenance when a vertex updates its distance and sends messages to its neighbors. Assume that at superstep $i-1$, y updates its distance and sends a message to x. Then, x at superstep i receives the message, updates its distance and sends a message to z. At superstep i, y sends again a message to x but this time x doesn't update its distance (at superstep $i+1$) and doesn't send a message to z. The provenance graph is seen in Figure 5.5b.

Ariadne employs a compact provenance graph representation. We observe that the provenance graph at superstep i contains a node for every input vertex that computed at superstep i. Moreover, it contains message edges for every edge in the input graph that was used to send/receive messages. Hence, the nodes and edges of the provenance graph that correspond to superstep i are a subset of the initial graph. Based on this observation, we propose a compact, relational representation of the provenance graph that utilizes the structure of the input graph, its vertices and edges, but not its values. Instead vertices are annotated with relations (tables) that contain the captured provenance information. So,

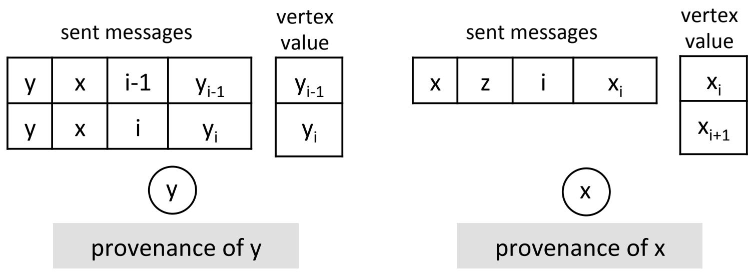


Figure 5.6. Compact provenance graph

in the compact representation the provenance graph contains all the input vertices and edges exactly once but with different data on them. The provenance relations annotating the nodes are:

- vertex-values(x,d,i): Value d of vertex v at superstep i.
- edge-values(x,y,d,i): Value d of edge between vertices x and y at superstep i.
- send-message(x,y,i,m): Message m sent from vertex x to its outgoing neighbor y at superstep i.
- receive-message(x,y,i,m): Message m received by vertex x from its incoming neighbor y at superstep i.

Figure 5.6 shows the compact representation of the provenance graph in Figure 5.5b. The send-message edges of the graph become tuples in the `send-message` relation of vertices x and y, and the receive-message edges become tuples in relation `receive-message` of vertex x. The evolution of y and x is captured by the tuples in relation `vertex-values` that has the value of each vertex at every superstep it was active.

Both formats contain the same information but they differ in their memory-requirements. If the provenance graph contains n instances of a vertex, the compact format contains one vertex with n tuples. It is easy to see that it is much cheaper to represent n data items (like numbers or strings) in memory rather than vertex objects.

Note that this is a specific property of provenance graphs, not applicable to general graphs, and is based on the intuition that nodes connected through evolution edges represent the same vertex at different instances in the computation and hence can be compacted.

5.4 Provenance querying

We argue that provenance can convey more information than derivation tracing given the correct tools. Providing a declarative high-level query language and a methodology to evaluate queries at scale offers new means of exploiting provenance. We already discussed the performance benefits Datalog offers for expressing graph analytics in Chapter 4. Since the input of graph analytics and their provenance is represented as graphs, Datalog is an excellent choice for a provenance query language. Moreover, this allows both graph analytics and provenance queries to be expressed in the same language.

In this section, we describe ARIADNE's query language and its properties and how we used DATALOGRAPHY to provide two novel evaluation methods allowing for efficient and scalable evaluation.

5.4.1 Provenance Query Language

Datalog queries in DATALOGRAPHY have access to the EDB predicates `vertices` and `edges` that describe the input graph. PQL queries need a richer vocabulary to express the annotations on provenance nodes and the different kinds of provenance edges. We introduce the EDB predicates shown in Table 5.1, populated by provenance capturing. Notice, how they correspond to the relations in the compact provenance graph representation of Section 5.3.

Now, we can show the Datalog notation for our running example:

The first rule creates table **eps** at a node x if the distance d_1 of vertex x at superstep i and its distance d_2 at the preceding superstep j differ less than a threshold ϵ. The next rule, **opt**, has two bodies: the first creates table **opt** at superstep i and initializes the

Table 5.1. Provenance EDB predicates

Provenance predicate	**Description**
superstep(x,i)	True if vertex x was active at superstep i
value(x,i,d)	True if vertex x was active at superstep i and had value d
evolution(x,i,j)	True if vertex x was active at supersteps i, j and i is the predecessor of j
send-message (x,y,i,m)	True if vertex x sent message m to vertex y at superstep i
receive-message (x,y,i,m)	True if vertex x received message y from vertex u at superstep i

$\mathbf{eps}(x,i) \leftarrow \text{value}(x,i,d_1), \text{value}(x,j,d_2), \text{evolution}(x,j,i), \text{udf-diff}(d_1,d_2,\epsilon).$
$\mathbf{opt}(x,0,i) \leftarrow \neg\, \text{eps}(x,i), \text{superstep}(x,i).$
$\mathbf{opt}(x,c+1,i) \leftarrow \text{opt}(x,c,j), \text{eps}(x,i), \text{evolution}(x,j,i).$
$\mathbf{max\text{-}opt}(x,MAX(c)) \leftarrow \text{opt}(x,c,i).$

Query 16. Running example: Tuning query

superstep counter (c) with 0 for every superstep a node x is not in **eps**. The second body increments c by 1 whenever x is in **eps**. Effectively, these two rules count the number of consecutive supersteps for which the distance of a vertex changed less than the threshold. When the streak of consecutive supersteps is broken, the counter resets to 0. Rule `max-opt` aggregates all the c values for a vertex and returns the maximum. In the end, a developer has the information of how many times the distance of a vertex changed less than ϵ and what is the longest streak of consecutive supersteps it occurred.

5.4.2 Layer-at-a-time evaluation

Ariadne uses PQL both for capturing and analyzing the provenance graph. Taking advantage of the structure of the provenance graph and the normal form of PQL, Ariadne offers scalable and efficient provenance management in which queries are evaluated *online* while the analytic is computing. Then, at the end of computation the results of the analytic and PQL queries exist.

We define a subclass of PQL, namely the *directed* queries, for which an ordering of the provenance graph is possible so that the queries are evaluated on "sliding windows" of the graph instead of its entirety. These sliding windows are layers defined as:

Definition 3 (Layers)**.** *Let G_{PR} be a provenance graph. Let n be the diameter of G_{PR} when captured for an analytic that computed for n supersteps. For $0 \leq i \leq n$ we inductively define the following family of layers:*

- *L_0 is the set of leaves of G_{PR},*
- *L_i is the set of leaves of $G_{PR} \setminus L_0 \cup \ldots \cup L_{i-1}$*

G_{PR} can be decomposed into $n+1$ such layers.

Consider again the provenance graph of Figure 5.5 created by capturing provenance for SSSP only this time both `send/receive message` edges are included. The layers of the graph can be seen in Figure 5.7. Assume node x_i in layer L_i evaluates rule

$$R_1(x,\ldots) \leftarrow T(y), \text{receive-message}(x,y,i,m), S(z), \text{send-message}(x,z,i,m)$$

The rule accesses remote predicate $\mathbf{T}(\mathbf{y})$ from a neighbor y that sent a message to x during computation (provenance node y_{i-1} in layer L_{i-1}) , and remote predicate $\mathbf{S}(\mathbf{z})$ from a neighbor z that x sent a message to (provenance node z_{i+1} in layer L_{i+1}). In order to evaluate rule R_1 at x_i, the message communication of Figure 5.8 must occur where node y_{i-1} sends relation $\mathbf{T}(\mathbf{y})$ and node z_{i+1} sends relation $\mathbf{T}(\mathbf{z})$ to x_i.

We can assign an *origin* to the remote predicates in a rule determined by whether their location specifier appears in a `receive-message` or `send-message` predicate. A remote predicate, like $\mathbf{T}(\mathbf{y}) \in$ *body*(R_1) has forward origin because its location specifier (y) appears in a `receive-message` predicate and , during query evaluation,node y in a lower layer sends relation T to x. On the other hand, remote predicate $\mathbf{T}(\mathbf{z}) \in$ *body*(R_1) has backwards origin as its location specifier (z) appears in a `send-message` predicate

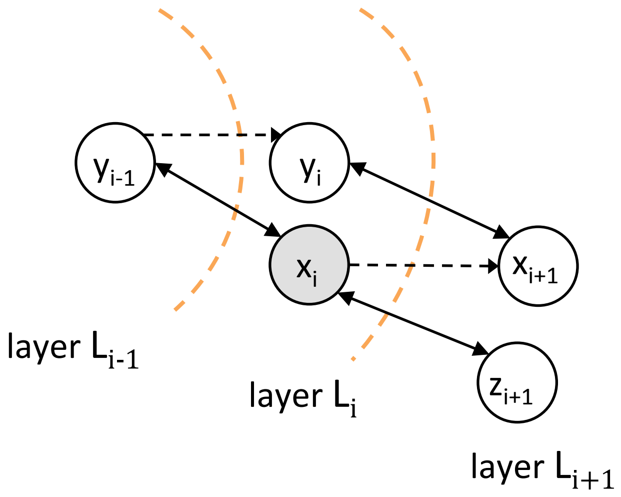


Figure 5.7. Provenance graph of SSSP divided into three layers.

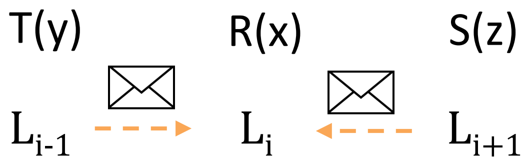


Figure 5.8. Information flow in evaluation of R_1

and neighbor z of a higher layer must sent T to x. It is clear, that evaluation of rule R_1 cannot impose an ordering on the layers of the provenance graph as its remote predicates have both forward and backward origin.

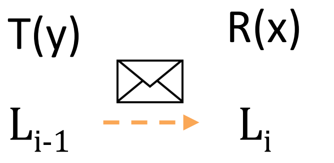
On the other hand, consider rule

$$R_2(x) \leftarrow T(y), \text{receive-message}(x, y, i, m)$$

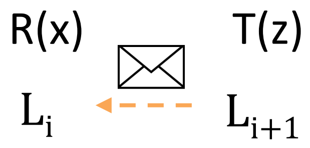
whose remote predicate is of forward origin. During query evaluation, node y in layer $i-1$ sends a message with relation **T** to x in layer i as seen in Figure 5.9a.

Additionally, consider rule

$$R_3(x) \leftarrow S(y), \text{send-message}(x, y, i, m)$$



(a) Message sending in rule R_2



(b) Message sending in rule R_3

Figure 5.9. Information flow in directed queries

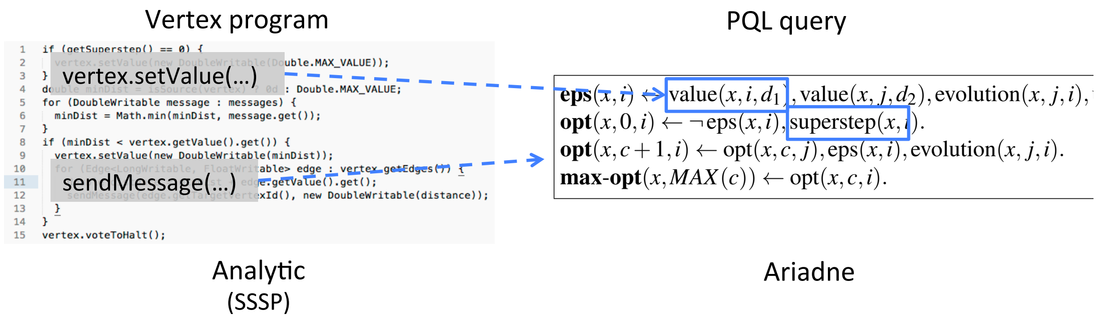
whose remote predicate is of backward origin. Figure 5.9b shows the message flow during query evaluation where a node z sends a message to x.

If all remote predicates in the rules of a PQL query are of the same origin, then it is possible to define an ordering on the layers of the provenance graph and evaluate queries in a *layer-at-a-time* manner. As such, all nodes in the same layer evaluate a query and send their query results to nodes in the next layer. The next layer, according to the ordering is a higher layer for forward queries or lower layer for backward queries. Then all nodes in the next layer evaluate the query and so forth. Based on this, we define the *directed* PQL queries as follows:

Definition 4 (Directed)**.** *A PQL query Q is directed if for every rule in Q the variables of remote predicates in their bodies appear in either send-message predicates or receive-message predicates but not both.*

A directed PQL query is *forward* if its remote predicates are of forward origin (like R_2) or *backward* if they are of backward origin (R_3). Query 16 from our running example is a forward PQL query.

Looking at the direction messages are exchanged during query evaluation and the order in which provenance layers are accessed, one realizes that forward query evaluation follows the same direction as analytic computation. Capitalizing on this, Ariadne evaluates forward queries *online* alongside the analytic. Ariadne intercepts the normal execution of an analytic, after all vertices have updated their values and created their



```
if (getSuperstep() == 0) {
  vertex.setValue(new DoubleWritable(Double.MAX_VALUE));
}
double minDist = isSource(vertex) ? 0d : Double.MAX_VALUE;
for (DoubleWritable message : messages) {
  minDist = Math.min(minDist, message.get());
}
if (minDist < vertex.getValue().get()) {
  vertex.setValue(new DoubleWritable(minDist));
  for (Edge<LongWritable, FloatWritable> edge : vertex.getEdges()) {
    [illegible]st + edge.getValue().get();
    sendMessage(edge.getTargetVertexId(), new DoubleWritable(distance));
  }
}
vertex.voteToHalt();
```

$$\begin{aligned}
&\textbf{eps}(x,i) \leftarrow \boxed{\mathrm{value}(x,i,d_1)}, \mathrm{value}(x,j,d_2), \mathrm{evolution}(x,j,i),\\
&\textbf{opt}(x,0,i) \leftarrow \neg\mathrm{eps}(x,i), \boxed{\mathrm{superstep}(x,i)}.\\
&\textbf{opt}(x,c+1,i) \leftarrow \mathrm{opt}(x,c,j), \mathrm{eps}(x,i), \mathrm{evolution}(x,j,i).\\
&\textbf{max-opt}(x,MAX(c)) \leftarrow \mathrm{opt}(x,c,i).
\end{aligned}$$

Figure 5.10. Vertex x at superstep i evaluates vertex program and PQL query.

messages to be sent, and uses this information as input to provenance query evaluation as seen in Figure 5.10. Online query evaluation incurs minimal overheads, ranging from 1.3x - 2x depending on the size of provenance information in the result of PQL queries.

Not all queries can be evaluated in an online fashion. Non-directed (like rule R_1) or backward (like rule R_3) PQL queries must be evaluated offline, after the analytic has terminated, since the order in which provenance layers are accessed does not follow the order of analytic computation. However, even in this case, Ariadne offers a scalable evaluation mode that does not require the entire provenance graph. As backward queries are directed, they can also be evaluated on a layer of the provenance graph at a time. The only difference with the online mode is that the layers are accessed in decreasing order of supersteps, starting from layer L_i and moving to layer L_{i-1}. Our experiments show that layer-at-a-time offline evaluation is scalable and more efficient than the naive approach of querying on the entire provenance graph.

Layer-at-a-time online and offline query evaluation are possible because the supersteps their evaluation requires are guarded by n, where n is the number of supersteps the analytic ran.

Lemma 1. *Evaluation of directed PQL queries requires at most n supersteps.*

Proof. Directed PQL queries impose a view on the provenance graph that is a DAG. The diameter of the DAG is n and its traversal requires at most n steps. □

For online PQL evaluation to be correct, we must ensure that query evaluation does not interfere with analytic computation and vice versa. Let us denote as $\text{Online}_{A,Q}(G)$ the lockstep evaluation of analytic A on input graph G and PQL query Q on the transient provenance information of A. The result of $\text{Online}_{A,Q}(G)$ contains both the result of A (modified vertex and edge values of G), as well as Q's results (new tables annotating the vertices of G). Correctness is ensured if the results of evaluating A by itself on G is the same as evaluating A in lockstep with Q and if the result of evaluating Q on the captured provenance of A is the same as evaluating Q using $\text{Online}_{A,Q}(G)$.

We define π_x to be the function that partitions the result of $\text{Online}_{A,Q}(G)$ into data (vertex and edge values) read/written by A or Q. Then,

Theorem 3. *Let A be a graph analytic, Q a forward PQL query, G the input graph and G_{PR} the provenance information of A. Then, $A(G) = \pi_A(\text{Online}_{A,Q}(G))$ and $Q(G_{PR}) = \pi_Q(\text{Online}_{A,Q}(G))$.*

Proof. It suffices to show that i) data modified by A are disjoint from the data modified by Q and ii) a vertex evaluating Q sends messages only when the vertex computing A sends messages and to the same neighbors.
i) A reads/writes vertex/edge data and messages. Q reads this data (by means of provenance) and appends tables resulting from query evaluation to the data and messages of a vertex. These tables are never accessed by A as it is agnostic to query evaluation.
ii) The definition of forward queries specifies that remote predicates in the body of a rule must be guarded by predicates `receive-message`. Hence, message exchange during query evaluation can only happen between vertices that exchanged messages during analytic computation. $\square$

5.5 Experimental evaluation

The main purpose of our evaluation is to investigate the effectiveness of Ariadne in declarative provenance management. Up to now, provenance had been captured imperatively and in its entirety whereas querying (limited to tracing) would occur offline. However, Ariadne enables a new methodology that tailors capturing to the user's needs via declarative queries. Moreover, instead of just capturing provenance, developers can use Ariadne to analyze the provenance while the analytic is computing. Finally, Ariadne can be used to query the captured provenance in an offline manner as well. As we will see below, materializing the entire provenance graph for offline querying is not scalable. Ariadne takes advantage of the characteristics of PQL and offers a stratified offline querying mode, materializing strata of the provenance graph, that is scalable and efficient.

The experiments were carried out on a cluster containing 7 Intel(R) Xeon(R) CPU E3-1270 v3 @ 3.50GHz machines, with 4 cores (2 hyper-threads per core), 32GB of RAM and 800GB of HDD. The operating system is Ubuntu 14.04, with jdk-1.0.7. We installed Hadoop 2.5 and Giraph 1.2. The datasets were all stored in HDFS with a replication factor of 2.

Algorithms and Datasets: We evaluate provenance capturing and querying on three graph algorithms that exhibit different computation and communication patterns: *PageRank* where all vertices execute at every superstep and send messages to all their outgoing neighbors, *SSSP* where the number of computing vertices starts out small (only the source), peaks and then reduces again and *WCC* where all vertices are active initially but drop as the computation proceeds. Note, we assigned random positive weights in the range of $0-1$ to the edges of the input graph for SSSP.

We used real-world datasets[1], namely indochina-2004 (IN-04), uk-2002 (UK-02), arabic (AR-05) and uk-2005 (UK-05). Their characteristics can be seen in Table 5.2. UK-2

[1]http://www.dis.uniroma1.it/challenge9/download.shtml

has the smallest degree, which as we will see below affects the runtime of both capturing and querying.

Table 5.2. Dataset characteristics

Dataset	V	E	Avg Degree	Avg Diameter
IN-04	7.4M	194M	26.17	28.12
UK-02	18.5M	298M	16.01	21.59
AR-05	22.7M	640M	28.14	22.39
UK-05	39.5M	936M	23.73	23.19
ML-20	16.5K	20M	121	1

Moreover, we experimented on the Alternating Least Squares (ALS) recommender algorithm using the MovieLens 20M[2] dataset with varying sizes of features (5-15). The user-movie ratings are represented as a bipartite graph, where an edge between user i and movie j carries a weight w indicating that user i gave the rating w to movie j. At every iteration, only one side of the bipartite graph computes, either the items or the movies since the algorithm optimizes the error function by fixing one set of variables and solving for the other. ALS converges when the error reaches an acceptable threshold. The ML-20 graph has 20M edges, 138493 users and 26744 movies. In the figures, for brevity, we use the notation ML-20^5, ML-20^{10} and ML-20^{15} for the variants of dataset ML-20 according to the number of features.

5.5.1 Capturing overheads

In traditional provenance management, the entire provenance graph is captured and stored for offline querying. We simulate this scenario with Query 17 that captures the full provenance graph and contains information about the vertex values at every superstep, and the messages sent and received. Table 5.3 shows the space overhead of the full provenance graph. For PageRank and SSSP the captured provenance is consistently $10x$ larger than

[2] http://grouplens.org/datasets/movielens/20m/

the input graph whereas for WCC it is $5x$.

$\textbf{prov-value}(x,i,v) \leftarrow \text{value}(x,i,v), \text{superstep}(x,i).$
$\textbf{prov-send}(x,y,i,m) \leftarrow \text{send-message}(x,y,i,m).$
$\textbf{prov-receive}(x,y,i,m) \leftarrow \text{receive-message}(x,y,i,m).$

Query 17. Capture full provenance graph

Table 5.3. Size comparison of input graph and full provenance graph

Dataset	Input	PageRank	SSSP	WCC
IN-04	4.1GB	45.1GB	42.7GB	22.6GB
UK-02	6.5GB	71GB	63.3GB	48.1GB
AR-05	13.8GB	148.1GB	118.6GB	76.4GB
UK-05	20.5GB	218.1GB	221.4GB	158.3GB

On the other hand, a declarative query language allows a user to tailor provenance capturing to her needs reducing this way space and time overheads. Consider for example Query 18 that captures information of what other vertices are influenced by a vertex in the input. This information is sufficient to answer the common provenance operation of forward tracing. We ran Query 18 for vertices that would reveal an upper bound for the overhead: for PageRank and WCC we chose the highest degree vertex whereas for SSSP we chose the source. Table 5.4 shows the size of the tailored provenance graph which in all three cases contains more than 80% of the input vertices.

$\textbf{prov}(x,i,v) \leftarrow \text{value}(x,i,v), \text{superstep}(x,i), x == \alpha, i == 0.$
$\textbf{prov}(x,i,v) \leftarrow \text{receive-message}(x,y,i,m,), \text{prov}(y,j), \text{value}(x,i,v).$

Query 18. Capture tailored provenance

Figure 5.11 shows the runtime overhead full and tailored provenance capturing incur. We see that the overhead of Query 17 ranges in $2.7x - 3.4x$ for PageRank and $3x - 5.6x$ for SSSP and WCC. The overhead of Query 18 is always lower than $2x$. When the provenance graph exceeds the size of available RAM, Ariadne offloads it asynchronously

Table 5.4. Size comparison of input graph and tailored provenance graph

Dataset	Input	PageRank	SSSP	WCC
IN-04	4.1GB	2.6GB	2.1GB	1.8GB
UK-02	6.5GB	3.5GB	2.9GB	2.5GB
AR-05	13.8GB	8GB	6.3GB	5.5GB
UK-05	20.5GB	13.9GB	14.3GB	8.4GB

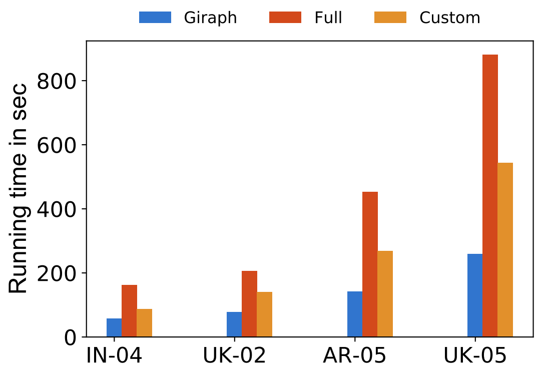
to HDFS. The rate of capturing and the rate of offloading play an important role in the scalability of the system. For ALS, for example, Ariadne could not capture the full provenance graph as the size of provenance for the smallest dataset (ML-20[5]), for one superstep, exceeded 80GB. Further research is required on how to support such sizes and generation rates of provenance information.

5.5.2 Online provenance querying

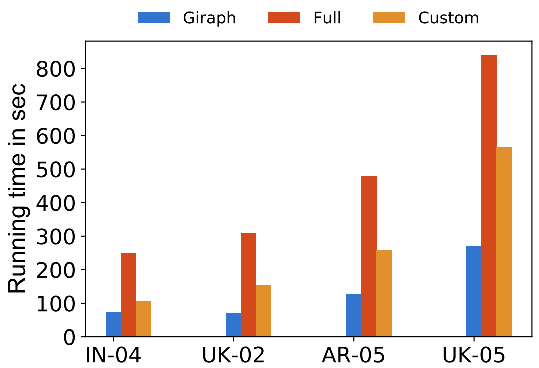
With Ariadne, a user can capture tailored provenance "on-demand": she can dynamically specify the vertices for which provenance is captured, in which circumstances and what information is part of the provenance. Moreover, she can issue provenance tracing operations on demand, only for the vertices identified by online queries. The same queries can be evaluated in an offline manner as well. Offline querying is useful when one does not know the queries up front. However, we see that the overhead of online provenance querying is so much lower than capturing-querying offline, that it is more performant to run the analytic multiple times, each with a different query.

In the figures below, we compare the running time of:

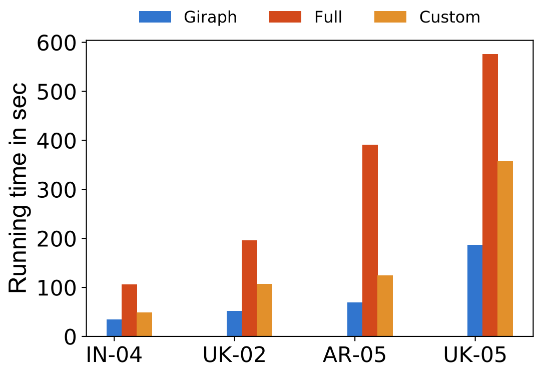
- A graph analytic running on Giraph without any provenance capturing or querying (`Giraph`).
- Online querying (`Online`).
- Offline querying in a layer-at-a-time manner (`Offline-L`).



(a) PageRank



(b) SSSP



(c) WCC

Figure 5.11. Runtime overhead of Query 17 and Query 18.

- Offline querying on the full provenance graph (`Offline`).

Note, the running times reported for offline querying do not include the capturing overheads.

Execution monitoring

Queries in this group specify conditions that should always be true for every vertex and every superstep. The provenance in the result of the queries contains instances where the condition fails. We envision these queries to be always "on", in the sense that they would be part of every run of an analytic. For that, it is imperative for them to incur minimal overhead.

We provide two queries for every analytic.

PageRank Query 19 checks that when the sum of the received messages of a vertex is greater than 0, then the in-degree of that vertex is greater than 0. In Giraph, messages can be sent to vertices by using their vertex ID. If the vertex ID is not that of an actual neighbor, a vertex without any incoming neighbors may receive a message erroneously. The runtime overhead in Figure 5.12a for Online is on average 1.14x whereas for *Offline-L* it is 3x and for *Offline* it is 4x.

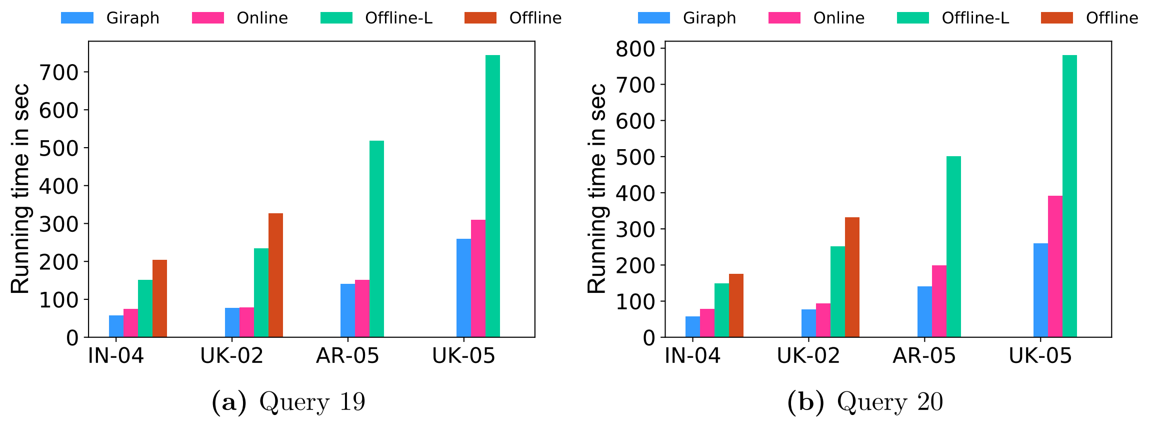
$\textbf{in-degree}(x, \text{COUNT}(y)) \leftarrow \text{edge}(y, x).$
$\textbf{check-failed}(x, y, i) \leftarrow \text{in-degree}(x, d), \text{receive-message}(x, y, i, m), d == 0.$

Query 19. PageRank Audit 1

Query 20 highlights a vertex as problematic, if its rank changed but the rank of its incoming neighbors hasn't changed. The query is parameterized by a threshold of how much change in rank a developer considers interesting. The query compares the old and new rank of a vertex and if their difference is larger than a threshold $\epsilon = 0.1$, it notifies the outgoing neighbors of that vertex. If none of the incoming neighbors changed (more than ϵ) but the rank of the vertex did, then that vertex is part of the result. The runtime overhead in Figure 5.12b is 1.4x using Online, 3.1x for *Offline-L*, and 3.7x for *Offline*.

big-change$(x,i) \leftarrow$ value(x,i,d_1), value(x,j,d_2), evolution(x,i,j), $|d_1 - d_2| > \epsilon$.
neighbor-change$(x,i) \leftarrow$ receive-message(x,y,i,m), big-change(y,i).
problem$(x,i) \leftarrow$ big-change(x,i), $\neg$neighbor-change(x,i)

Query 20. PageRank Audit 2



(a) Query 19 **(b)** Query 20

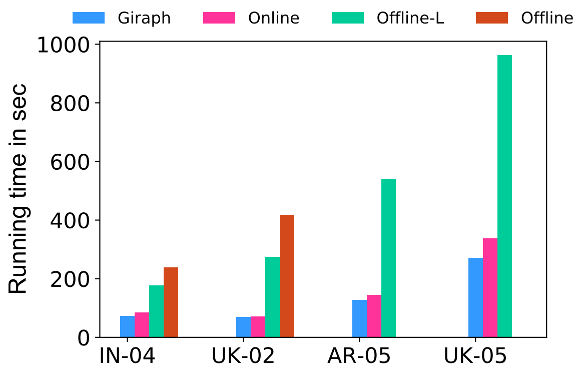
Figure 5.12. Running time of PageRank audit queries.

SSSP and WCC Query 21 checks that when a vertex updates its value, it is because it received messages and because its new value is smaller than the previous one. SSSP and WCC work under the assumption of positive weights and positive vertex IDs respectively. If the input is corrupted, such as if there is an edge with negative weight, or the algorithm assigns the wrong label, the query will highlight it. Figure 5.13a shows the runtime for SSSP: The overhead of Online is on average 1.13x whereas for *Offline-L* it is 3.5x and for *Offline* it is 4.6x. The running time overheads for WCC in Figure 5.13c are similar.

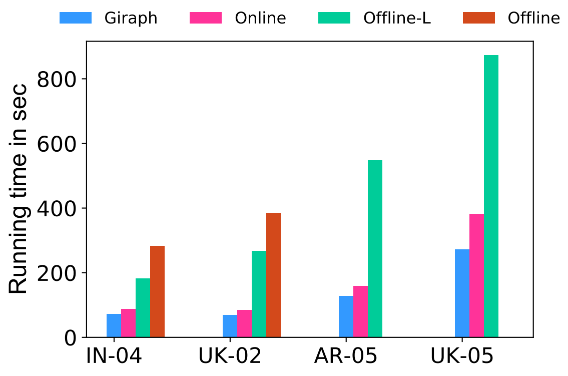
check-failed$(x,i) \leftarrow$ value(x,i,d_1), value(x,j,d_2), evolution(x,i,j),
    receive-message(x,y,i,m), $d_2 \leq d_1$.

Query 21. SSSP and WCC Audit 1

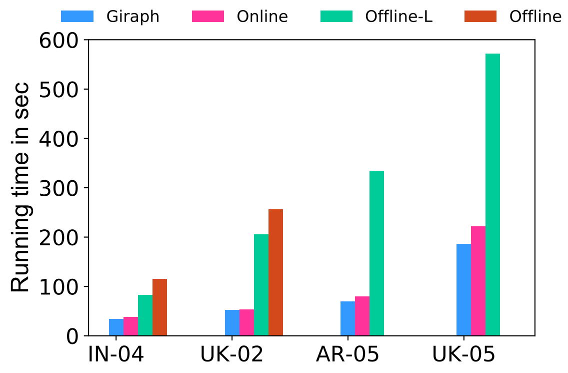
For SSSP and WCC, a vertex updates its distance or label only when it receives a message from an incoming neighbor with a smaller distance or label. Query 22 ensures that if a vertex did not receive a message from any of its neighbors, then its value doesn't change. Figure 5.13b shows the overhead for SSSP is on average 1.3x for Online whereas



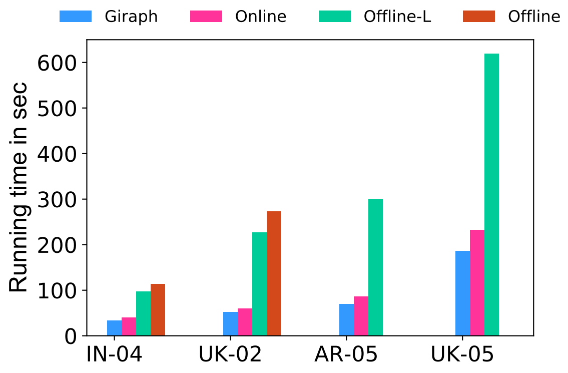
(a) SSSP Query 21



(b) SSSP 22



(c) WCC 21



(d) WCC 22

Figure 5.13. Running time of SSSP and WCC audit queries.

for *Offline* it is $4.7x$ and for *Offline-L* it is $3.6x$. In Figure 5.13d the overhead for WCC and Online is $1.2x$, for *Offline-L* it is $3.7x$ and *Offline* it is 4.3.

$\textbf{neighbor-change}(x,i) \leftarrow \text{receive-message}(x,y,i,m).$
$\textbf{problem}(x,i) \leftarrow \text{value}(x,i,d_1), \text{value}(x,j,d_2), \text{evolution}(x,i,j), \neg\text{neighbor-change}(x,i),$
$\quad d_1 \neq d_2.$

Query 22. SSSP and WCC Audit 2

ALS Query 23 checks that the local error for every vertex is between the range of $0-5$ which is the range of the ratings in the input file. The error is computed by subtracting the actual rating from the predicted one during every superstep of the computation. The query identifies, when a vertex fails the check, if it is because the input file contains a rating outside of the expected range $(0-5)$, or because the prediction computed at a superstep is outside the range. Figure 5.14a shows the runtime overhead for this query is $1.04x$ for Online.

input-failed$(x,y,i) \leftarrow$ prov-error(x,y,i,e), edge-value$(x,y,i,w), e < 0, e > 5, w < 0, w > 5.$
algo-failed$(x,y,i) \leftarrow$ prov-error(x,y,i,e), prov-prediction$(x,y,i,p), e < 0, e > 5, p < 0,$
$p > 5.$

Query 23. ALS Audit 1

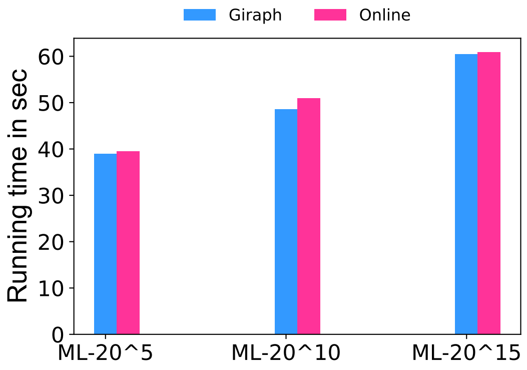
Query 24 identifies users or items whose average error in rating prediction increases in consecutive supersteps. It first computes the local average error per vertex and superstep by summing the errors across all its neighbors and dividing by the out-degree. It then compares the average error for two consecutive supersteps and checks that the error has not increased more than a threshold. For a threshold of 0.5, the query returns 30% of the vertices indicating that their error has increased. Finding such vertices is useful as it can indicate that these vertices converge to a wrong solution and should be handled differently by the algorithm. Figure 5.14b shows the runtime overhead for this query is $1.2x$ for Online.

degree$(x, COUNT(y)) \leftarrow$ receive-message(x,y,i,m).
sum-error$(x,i,SUM(e)) \leftarrow$ prov-error(x,y,i,e).
avg-error$(x,i,s/d) \leftarrow$ sum-error(x,s), degree(x,d).
problem$(x,e_1,e_2,i) \leftarrow$ avg-error(x,i,e_1), avg-error(x,j,e_2), evolution$(x,i,j), e_1 > e_2 + \epsilon$

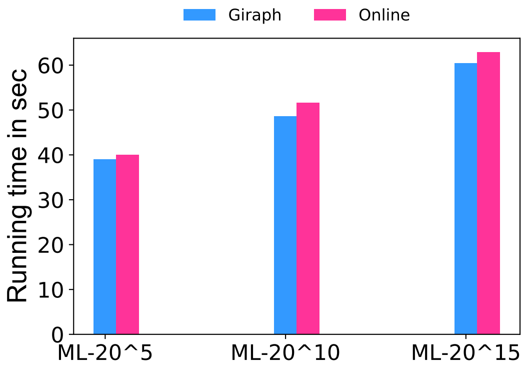
Query 24. ALS Audit 2

Performance tuning

We now turn to a novel idea of provenance usage that assists developers in tuning their algorithms. We measure the overhead of our running example Query 16, copied here for ease of readability. Moreover, we show that insights gained from using the query on one dataset, are transferable to unseen datasets.



(a) Query 23



(b) Query 24

Figure 5.14. Running time for ALS audit queries.

We use the same query for all our analytics by parameterizing it with different thresholds and vertex value comparison functions. For instance for PageRank, SSSP and WCC the query subtracts the previous and new vertex value whereas for ALS it compares their euclidean distance. At the end of computation, the query results contain the provenance information: i) at what supersteps did a vertex value change less than a threshold and ii) The number of *consecutive* supersteps a vertex is a candidate for optimization. The larger the fraction of vertices whose value doesn't change and the longer the length of consecutive supersteps is, the more optimization is achieved.

We measure the error of approximation in the same manner as [63] by using the L_p norm of a vector v defined as: $L_p(v) = (\sum_{i=1}^{n} {v_i}^p)^{\frac{1}{p}}$. Let r_0 be the vector of results of the original analytic and r_1 the vector of results for the optimized analytic. Then the normalized error is: $L_p(r_0 - r_1)/L_p(r_0)$. We choose the correct error definition based on the characteristics of the data and algorithms.

$\mathbf{eps}(x,i) \leftarrow \text{value}(x,i,d_1), \text{value}(x,j,d_2), \text{evolution}(x,j,i), \text{udf-diff}(d_1,d_2,\epsilon).$
$\mathbf{opt}(x,0,i) \leftarrow \neg\text{eps}(x,i), \text{superstep}(x,i).$
$\mathbf{opt}(x,n+1,i) \leftarrow \text{opt}(x,n,j), \text{eps}(x,i), \text{evolution}(x,j,i)$

Running example 16

Figure 5.16 reports the runtime of the optimization query for all datasets. However, we analyzed the results only for UK-02 and based on the findings on this dataset, we applied the optimization with the *same* threshold to the other datasets to see if it is applicable on unseen graphs.

For PageRank $\epsilon = 0.01$, the query finds that for 60% of the vertices their rank doesn't change in 10 out of 20 supersteps. This is a very strong result and since the optimization is already part of some PageRank implementation, it highlights how useful this query is. The error when computing the optimized PageRank is shown in Table5.5 and ranges between 10^{-3} to 10^{-5}. The table also shows the median of the ranks for the initial analytic and the optimized analytic as a means of comparison with the error. The speedup of the optimized version in Figure 5.15a is $1.4x$. The overhead of the query, shown in Figure 5.16a is $1.3x$ for Online, $3.2x$ for *Offline-L* and 3.8 for *Offline*.

For SSSP and $\epsilon = 0.1$ the query result contains 87% of the input vertices with 11% of them not changing their value for more than 10 consecutive supersteps. The overhead of the query is $1.5x$ on average for Online, $3.5x$ for *Offline-L* and $5x$ for *Offline*. Again the results lead a developer to incorporate the optimization to SSSP (to get the code in Figure 5.16b). Table 5.6 shows the error across all datasets is 10^{-2} when using the same threshold. The runtime improvements, see Figure 5.15b, are $1.8x$ over the baseline.

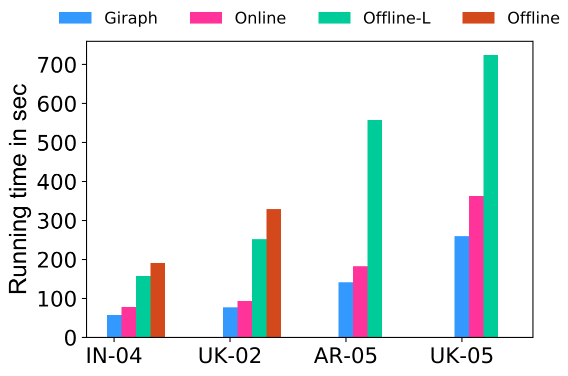
Table 5.5. PageRank: Relative error (L_2) for $\epsilon = 0.01$ and median values of original (A) and optimized (B) analytics.

Dataset	**Error**	**Median A**	**Median B**
IN-04	10^{-3}	0.18	0.16
UK-02	10^{-3}	0.2	0.17
AR-05	10^{-4}	0.18	0.15
UK-05	10^{-5}	0.2	0.17

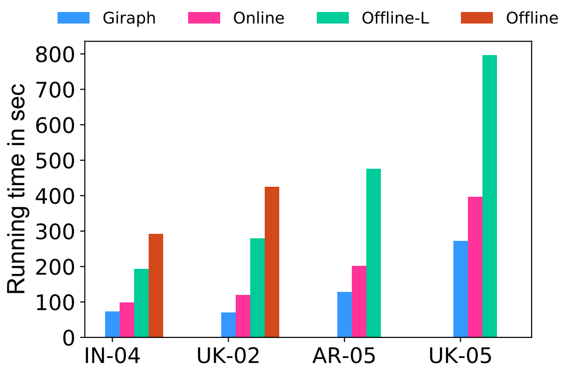
We don't expect this idea of optimization to work with WCC as it's values are categorical and cannot be approximated this way. We used a threshold of 1 for the query

Table 5.6. SSSP: Relative error (L_1) for $\epsilon = 0.1$ and median values of original (A) and optimized (B) analytics.

Dataset	Error	Median A	Median B
IN-04	10^{-2}	5	5.2
UK-02	10^{-2}	4.4	4.5
AR-05	10^{-2}	5.6	5.8
UK-05	10^{-2}	3.7	3.8



(a) PageRank

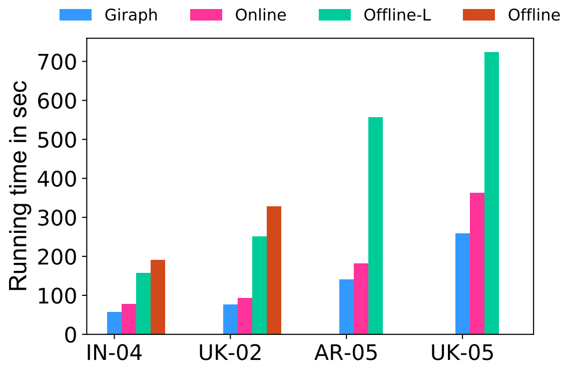


(b) SSSP

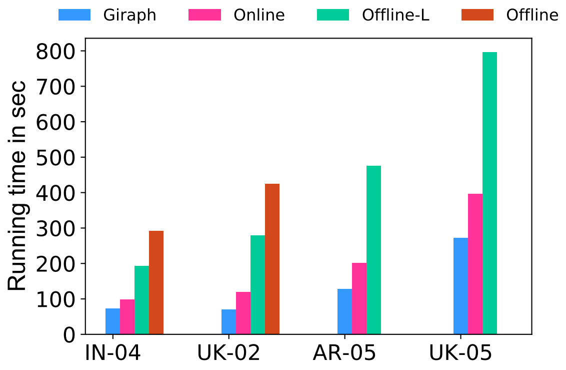
Figure 5.15. Runtime improvement between original (Giraph) and optimized (Giraph - opt) analytic

since the amount of difference between component IDs doesn't matter, but rather the fact that vertices are assigned to different components. The query finds that 60% of the vertices are an optimization candidate at some superstep but never for a consecutive streak. This result already proves a developer should not pursue the optimization. Sure enough, when running the "optimized" version, the normalized relative error is 0.9 across all datasets. The overhead of evaluating the query, in Figure 5.16c using Online is 1.6x, for *Offline* is 5 and 3.6x for *Offline-L*.

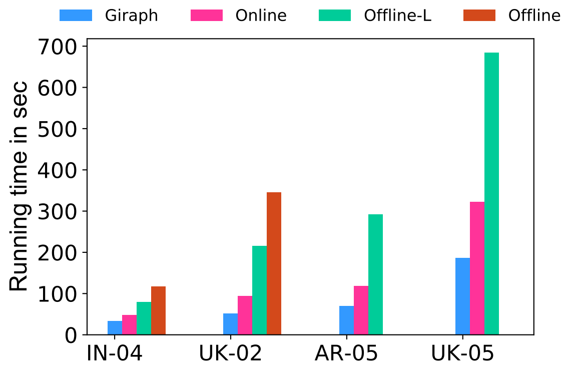
Finally, for ALS the query returns too few vertices (optimization candidates) to pursue this optimization since it requires a more fine-tuned convergence criterion. As future work, we plan to do a user study with developers using ARIADNE to write algorithm-aware



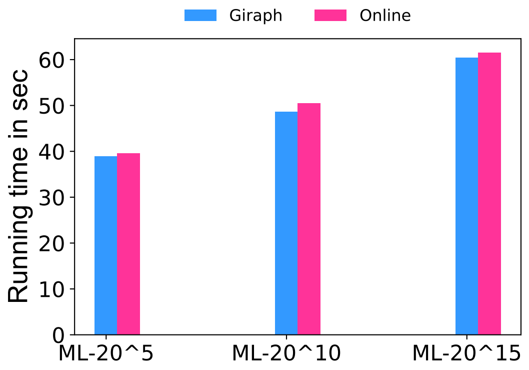
(a) PageRank



(b) SSSP



(c) WCC



(d) ALS

Figure 5.16. Running time of running example Query 16

tuning queries. Nevertheless, the running time overhead for ALS is in Figure 5.16d and is always lower than $1x$.

5.5.3 Offline provenance querying

Tracing the provenance backward (with respect to the direction the computation unfolds) to identify the input data items that lead to an item in the output is a common operation. Ariadne supports this kind of reasoning via offline querying. Below we compare two approaches: i) Capture the entire provenance graph and perform backward tracing using Query 26 and ii) Capture a tailored provenance graph using Query 27 and trace it using Query 28. In both cases, the tracing query specifies the starting vertex α and superstep σ. Relation **back-lineage** contains the vertices the trace reaches at superstep 0.

start$(x,i) \leftarrow$ superstep$(x,i), i = \sigma, x = \alpha$.
back-trace$(x,i) \leftarrow$ start(x,i).
back-trace$(x,i) \leftarrow$ prov-send(x,y,i,m), back-trace$(y,j), j = i+1$.
back-lineage$(x,d) \leftarrow$ back-trace(x,i), prov-value$(x,i,d), i = 0$.

Query 26. Backward lineage on full provenance graph

Query 26 traces the provenance graph using `send-message` edges. Notice how the values of messages are never accessed. Using ARIADNE, users can customize capturing and disable capturing of message values as they are not needed. Moreover, for analytics that send messages to all their outgoing neighbors (instead of a dynamic subset) it is not necessary to capture the `send-message` edges that replicate the information of which neighbor a message is sent to across supersteps. Instead, a user can specify to capture only the **edges** EDB relation of the input graph that contains the same information.

Query 27 captures the tailored provenance for backward tracing. Rule `prov-value` captures the value of a vertex at every superstep. Rule `prov-send` captures the superstep at which a vertex sends messages and rule `prov-edges` captures the outgoing edges of a vertex. The tailored backward tracing Query 28 differs from Query 26 in using relation **edges** instead of **send-message**.

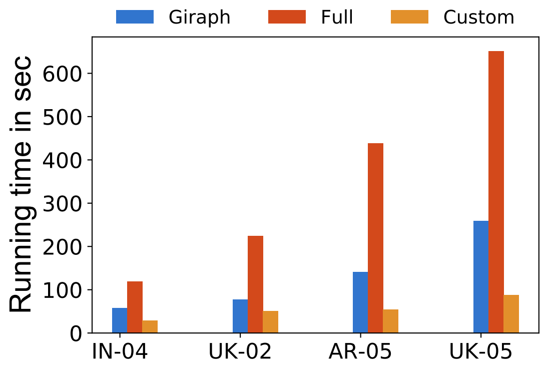
prov-value$(x,i,v) \leftarrow$ value(x,i,d), superstep(x,i).
prov-send$(x,i) \leftarrow$ send-message$(x,y,i,m,)$.
prov-edges$(x,y) \leftarrow$ edges(x,y).

Query 27. Capture tailored provenance for backward query

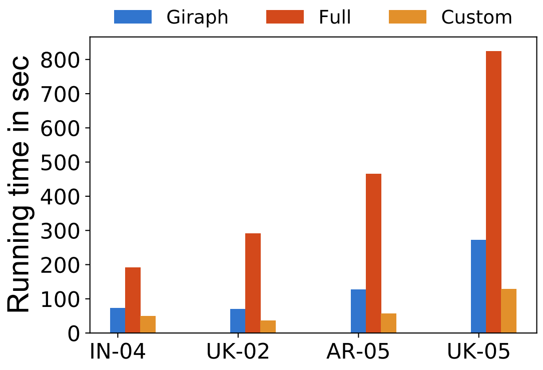
start$(x,i) \leftarrow$ prov-value$(x,i,v), i = \sigma, x = \alpha$.
back-trace$(x,i) \leftarrow$ start(x,i).
back-trace$(x,i) \leftarrow$ prov-edges(x,y), prov-send(x,i), back-trace$(y,j), j = i+1$.
back-lineage$(x,d) \leftarrow$ back-trace(x,i), prov-value$(x,i,d), i = 0$.

Query 28. Backward lineage on tailored provenance graph

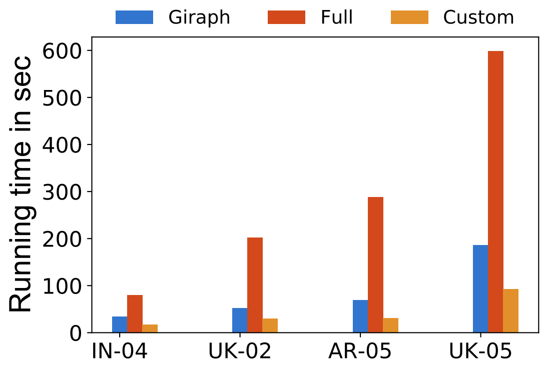
We compare the runtime of layer-at-a-time offline evaluation using the full prove-



(a) PageRank



(b) SSSP



(c) WCC

Figure 5.17. Runtime overhead for backward lineage query 26

nance graph (*Full*) against using the tailored provenance graph(*Custom*). The times don't include capturing. Moreover, we include the analytic's time (Giraph) for reference. We started the trace from a vertex that computed in the last superstep of the analytic and traversed the provenance graph to superstep 0.

In all cases, *Custom*'s runtime is 50% of the runtime of the base analytic. For PageRank the overhead for *Full* is 2.6x whereas for *Custom* it is 0.5x. For SSSP, the overhead of *Full* is 3.4x and for *Custom* it is 0.5x. For WCC the overheads are similar. This highlights the strength of tailored capturing showing can one can take advantage of the characteristics of her analytic and provenance queries to reduce the size of captured provenance and cut down significantly the querying time.

5.6 Related work

Although provenance querying has been studied in the context of databases [29, 40], scientific workflows [21, 8] and large-scale distributed engines [38, 18], there is no work addressing provenance in a setting where the data model of the computation and the data model of the provenance are both graphs. Moreover, with the exception of [29], no previous work addresses online provenance querying while the computation that produces the provenance unfolds.

Focusing on approaches on large-scale distributed engines, their limitations can be summarized in: i) Provenance is captured imperatively, behind the scenes. This does not allow a developer to customize what information is included in the captured provenance. ii) Provenance can be used only offline. iii) Only imperative tracing of provenance is offered.

Graft [60], is the only other tool addressing debugging in VC systems. Users can imperatively specify a small set of vertices (less than 10) to visualize and analyze using a step-wise debugger *locally*. Although, visualization is helpful in understanding graph algorithms, users need help in identifying on which vertices to narrow down. Moreover, debugging the computation logic locally gives no guarantee as to whether the fixes will translate into a fix at scale.

Provenance on batch-processing system was first addressed by Newt [44] and Ramp [55] that capture provenance for MapReduce workflows on Hadoop [2] and offer offline tracing using external tools. Titian [38] instruments Spark with provenance capturing and is the only other large-scale system that allows provenance tracing using the same language the analytics are expressed. Titian manages to incur an overhead of $1.4x$ over the base runtime, much smaller to ours, due to three reasons: i) The provenance size of batch processing analytics is smaller than the size of the input ($30\% - 50\%$) as compared to $10x$ size for graph analytics ii) the workflows they experimented on have two stages compared to the $20-200$ supersteps of our graph analytics iii) Spark offers a built-in intermediary storage mechanism and Titian offloads to disk only when provenance does not fit in memory taking advantage of native Spark tools. As future work, we plan to look into out-of-core graph processing systems [64, 59, 83] to improve Ariadne's performance when capturing the full provenance graph.

[7] tracks How-Provenance [32] for Pig Latin operators. How-provenance is more expressive than lineage as it conveys not only what input items contributed to the computation of an output but also how. Like us, they model provenance as a graph that however, must be built through an offline process. A separate module allows querying of the provenance graph in the limited form of graph transformations such as zoom-in/out and deletion propagation (tracing).

[18] addresses backward provenance tracing on a differential dataflow system [52] for iterative analytics. A common problem with backward tracing is that the input data items returned are too many to be useful. The authors propose interesting ideas to prune the tracing size such as considering the time data items were produced and exploiting characteristics of algorithms like WCC that follow a top-k logic where only top-k input items are necessary to explain outputs. Using Ariadne, a developer is able to apply provenance pruning both to capturing (capture only the top-k data items) and to querying by customizing her queries.

5.7 Chapter summary

This chapter presented Ariadne, a first approach in capturing and querying provenance on large-scale graph analytics. We show that provenance can be used in more scenarios than traditional debugging given the right tools. Ariadne offers a high-level query language, PQL, that developers can use to mine provenance, without the need of capturing it. They can pose queries that validate invariants on the data and computation to ensure correct execution, queries that identify corner cases of an algorithm or outliers in the data, and queries that investigate the runtime behavior of fixpoint graph analytics. We identified a class of queries, namely *directed*, that can be evaluated on layers of the provenance graph instead of its entirety. Experiments showed that layer-at-a-time evaluation is more scalable and efficient than the straightforward approach of materializing the entire provenance graph. Finally, for a sublcass of directed queries, we presented a novel *online* evaluation mode that obviates the need for provenance capturing. Online evaluation shortcuts the traditional provenance querying approach by evaluation provenance queries while a graph analytic is executing. Our experiments showed that online evaluation incurs an average overhead of 1.3x.

This chapter, in full, has been submitted for publication of the material. Vicky Papavasileiou, Ken Yocum, Alin Deutsch. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Conclusion and Future Directions

In this dissertation we presented a declarative framework that supports the entire life-cycle of Big Graph analytics, from developing to tuning their behavior. First, we presented Datalography, a Datalog compiler and evaluation engine for *Vertex-Centric* systems. We showed our theoretical work on rewriting general Datalog queries to VC-Datalog queries that follow the *Vertex-Centric* paradigm thus amenable to efficient distributed evaluation. We designed and implemented optimizations in the form of logical rewritings that are portable to other evaluation frameworks. More specifically, we defined the logical abstraction of super-vertices that enable set-at-a-time evaluation in a transparent to the user way. Then, we presented the optimization of source-side combiners enabled by super-vertices. Our experiments showed that the concise, declarative formulation of Big Graph analytics on Datalography outperforms imperative code by a factor of $2.3x$-$7.8x$ (depending on the dataset and algorithm) over Apache Giraph whereas it achieves a speed-up of $1.4x$-$2.5x$ when compared to a custom partition-aware implementation of Apache Giraph.

Then, we proposed Ariadne a declarative approach for provenance management of Big Graph analytics on *Vertex-Centric* engines. We showed how a declarative provenance language (PQL) enables users to tailor provenance capturing and querying to their needs cutting down significantly on space and time overheads. Moreover, we introduced a novel provenance querying methodology that enables *online* provenance querying while a Big Graph analytic is executing so that the end of execution both the results of the analytic and the provenance query exist. Finally, we presented novel provenance queries that go beyond the traditional forward and backward tracing and allow developers to audit and fine-tune the behavior of their graph analytics. In our experiments with different graph analytics and input graphs, the overhead of online querying is on average $1.3x$ versus $8x$ of the traditional approach (capture everything, query offline).

A declarative framework for authoring and fine-tuning Big Graph analytics enables exciting opportunities for further research. One first direction is to investigate multi-query optimization techniques when both analytic and provenance queries are expressed in Datalog. Moreover, the high generation rate of provenance information as well as its size, incur large overheads when provenance is captured. One possible future direction is to investigate online provenance graph compression techniques as well as out-of-core graph processing. Lastly, we envision our declarative framework to play the role of a mediator over *Vertex-Centric* systems and graph databases, providing a common declarative abstraction. Such mediator is useful to identify the parts of a graph analytic that are best evaluated by an in-memory system compared to a database and vice-versa.

Bibliography

[1] Giraph. https://giraph.apache.org/.

[2] Hadoop. http://hadoop.apache.org.

[3] Parmetis - parallel graph partitioning and fill-reducing matrix ordering. http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview. Accessed: 2016-05-24.

[4] Foto N. Afrati and Jeffrey D. Ullman. Transitive closure and recursive datalog implemented on clusters. In *EDBT*, pages 132–143, 2012.

[5] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *EuroSys*, pages 223–236, 2010.

[6] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in bloom: a CALM and collected approach. In *CIDR*, 2011.

[7] Yael Amsterdamer, Susan B. Davidson, Daniel Deutch, Tova Milo, Julia Stoyanovich, and Val Tannen. Putting lipstick on pig: Enabling database-style workflow provenance. *PVLDB*, 5(4):346–357, 2011.

[8] Manish Kumar Anand, Shawn Bowers, and Bertram Ludäscher. Techniques for efficiently querying scientific workflow provenance graphs. In *EDBT*, volume 10, pages 287–298, 2010.

[9] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the logicblox system. In *SIGMOD*, 2015.

[10] François Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *SIGMOD*, pages 16–52, 1986.

[11] Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, pages 1151–1162, 2011.

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

[13] Yingyi Bu, Vinayak R. Borkar, Michael J. Carey, Joshua Rosen, Neoklis Polyzotis, Tyson Condie, Markus Weimer, and Raghu Ramakrishnan. Scaling datalog for machine learning on big data. *CoRR*, abs/1203.0160, 2012.

[14] Yingyi Bu, Vinayak R. Borkar, Jianfeng Jia, Michael J. Carey, and Tyson Condie. Pregelix: Big(ger) graph analytics on a dataflow engine. *PVLDB*, 8(2):161–172, 2014.

[15] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.

[16] Brian Chin, Daniel von Dincklage, Vuk Ercegovac, Peter Hawkins, Mark S. Miller, Franz Josef Och, Christopher Olston, and Fernando Pereira. Yedalog: Exploring knowledge at scale. In *SNAPL*, 2015.

[17] Avery Ching. Scaling apache giraph to a trillion edges. *Facebook Engineering blog*, 2013.

[18] Zaheer Chothia, John Liagouris, Frank McSherry, and Timothy Roscoe. Explaining outputs in modern data analytics. *PVLDB*, 9(12):1137–1148, 2016.

[19] Mariano P. Consens and Alberto O. Mendelzon. Low complexity aggregation in graphlog and datalog. *Theor. Comput. Sci.*, 116(1):95–116, 1993.

[20] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and lattices for distributed programming. In *SOCC*, page 1, 2012.

[21] Yingwei Cui and Jennifer Widom. Lineage tracing for general data warehouse transformations. *PVLDB*, 12(1):41–58, 2003.

[22] Jason Eisner and Nathaniel Wesley Filardo. Dyna: Extending datalog for modern AI. In *Datalog Reloaded*, pages 181–220, 2010.

[23] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey C. Fox. Twister: a runtime for iterative mapreduce. In *HDPC*, pages 810–818, 2010.

[24] Benedikt Elser and Alberto Montresor. An evaluation study of bigdata frameworks for graph processing. In *IEEE Big Data*, pages 60–67, 2013.

[25] Sumit Ganguly, Sergio Greco, and Carlo Zaniolo. Minimum and maximum predicates in logic programming. In *PODS*, pages 154–163, 1991.

[26] Allen Van Gelder. The well-founded semantics of aggregation. In *PODS*, pages 127–138, 1992.

[27] Allen Van Gelder. Foundations of aggregation in deductive databases. In *DOOD*, pages 13–34, 1993.

[28] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080, 1988.

[29] Boris Glavic and Gustavo Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE*, pages 174–185, 2009.

[30] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[31] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

[32] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *SIGMOD*, pages 31–40, 2007.

[33] Muhammad Ali Gulzar, Matteo Interlandi, Tyson Condie, and Miryung Kim. Debugging big data analytics in spark with *BigDebug*. In *SIGMOD*, pages 1627–1630, 2017.

[34] Yong Guo, Marcin Biczak, Ana Lucia Varbanescu, Alexandru Iosup, Claudio Martella, and Theodore L. Willke. How well do graph-processing platforms perform? an empirical performance evaluation and analysis. In *IPDPS*, pages 395–404, 2014.

[35] Minyang Han and Khuzaima Daudjee. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *PVLDB*, 8(9):950–961, 2015.

[36] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Özsu, Xingfang Wang, and Tianqi Jin. An experimental comparison of pregel-like graph processing systems. *PVLDB*, 7(12):1047–1058, 2014.

[37] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: an interactive tutorial. In *SIGMOD*, pages 1213–1216, 2011.

[38] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. Titian: Data provenance support in spark. *PVLDB*, 9(3):216–227, 2015.

[39] Martin Junghanns, André Petermann, Martin Neumann, and Erhard Rahm. Management and analysis of big graph data: Current systems and open challenges. In *Handbook of Big Data Technologies*, pages 457–505. 2017.

[40] Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Querying data provenance. In *SIGMOD*, pages 951–962, 2010.

[41] David B. Kemp and Kotagiri Ramamohanarao. Efficient recursive aggregation and negation in deductive databases. *IEEE Trans. Knowl. Data Eng.*, 10(5):727–745, 1998.

[42] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *EuroSys*, pages 169–182, 2013.

[43] Donald Kossmann. The state of the art in distributed query processing. *ACM CSUR*, 32(4):422–469, 2000.

[44] Dionysios Logothetis, Soumyarupa De, and Kenneth Yocum. Scalable lineage capture for debugging DISC analytics. In *SOCC*, 2013.

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

[46] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[47] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. Large-scale distributed graph computing systems: An experimental evaluation. *PVLDB*, 8(3):281–292, 2014.

[48] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[49] William R. Marczak, Shan Shan Huang, Martin Bravenboer, Micah Sherr, Boon Thau Loo, and Molham Aref. Secureblox: customizable secure distributed data processing. In *SIGMOD*, pages 723–734, 2010.

[50] Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. A declarative extension of horn clauses, and its significance for datalog and its applications. *TPLP*, 13(4-5):609–623, 2013.

[51] Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. Extending the power of datalog recursion. *VLDB J.*, 22(4):471–493, 2013.

[52] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *CIDR*, 2013.

[53] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *VLDB*, pages 264–277, 1990.

[54] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, 2011.

[55] Hyunjung Park, Robert Ikeda, and Jennifer Widom. RAMP: A system for capturing and tracing provenance in mapreduce workflows. *PVLDB*, 4(12):1351–1354, 2011.

[56] Teodor C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1988.

[57] Kenneth A. Ross. Modular stratification and magic sets for datalog programs with negation. *J. ACM*, 41(6):1216–1266, 1994.

[58] Kenneth A. Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. In *PODS*, 1992.

[59] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488. ACM, 2013.

[60] Semih Salihoglu, Jaeho Shin, Vikesh Khanna, Ba Quan Truong, and Jennifer Widom. Graft: A debugging tool for apache giraph. In *SIGMOD*, pages 1403–1408, 2015.

[61] Semih Salihoglu and Jennifer Widom. GPS: a graph processing system. In *SSDBM*, 2013.

[62] Jiwon Seo, Stephen Guo, and Monica S. Lam. Socialite: Datalog extensions for efficient social network analysis. In *ICDE*, 2013.

[63] Zechao Shang and Jeffrey Xu Yu. Auto-approximation of graph computing. *PVLDB*, 7(14):1833–1844, 2014.

[64] Zhiyuan Shao, Jian He, Huiming Lv, and Hai Jin. Fog: A fast out-of-core graph processing framework. *International Journal of Parallel Programming*, 45(6):1259–1272, 2017.

[65] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In *SIGMOD*, pages 1135–1149, 2016.

[66] Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. Optimizing recursive queries with monotonic aggregates in deals. In *ICDE*, pages 867–878, 2015.

[67] Yogesh Simmhan, Alok Gautam Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi S. Raghavendra, and Viktor K. Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. In *Euro-Par*, pages 451–462, 2014.

[68] S. Sudarshan and Raghu Ramakrishnan. Aggregation and relevance in deductive databases. In *VLDB*, pages 501–511, 1991.

[69] Narayanan Sundaram, Nadathur Satish, Md. Mostofa Ali Patwary, Subramanya Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *PVLDB*, 8(11):1214–1225, 2015.

[70] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From "think like a vertex" to "think like a graph". *PVLDB*, 7(3):193–204, 2013.

[71] Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *PVLDB*, 8(12):1542–1553, 2015.

[72] Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. Big graph analytics platforms. *Foundations and Trends in Databases*, 7(1-2):1–195, 2017.

[73] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *WWW*, pages 1307–1317, 2015.

[74] Weipeng P. Yan and Per-Ake Larson. Eager aggregation and lazy aggregation. In *VLDB*, 1995.

[75] Mohan Yang and Carlo Zaniolo. Main memory evaluation of recursive queries on multicore machines. In *IEEE Big Data*, pages 251–260, 2014.

[76] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

[77] Carlo Zaniolo, Natraj Arni, and KayLiang Ong. Negation and aggregates in recursive rules: the LDL++ approach. In *DOOD*, pages 204–221, 1993.

[78] Carlo Zaniolo, Mohan Yang, Ariyam Das, Alexander Shkapsky, Tyson Condie, and Matteo Interlandi. Fixpoint semantics and optimization of recursive datalog programs with aggregates. *TPLP*, 17(5-6):1048–1065, 2017.

[79] Carlo Zaniolo, Mohan Yang, Matteo Interlandi, Ariyam Das, Alexander Shkapsky, and Tyson Condie. Declarative bigdata algorithms via aggregates and relational database dependencies. In *AMW*, 2018.

[80] Qizhen Zhang, Hongzhi Chen, Da Yan, James Cheng, Boon Thau Loo, and Purushotham Bangalore. Architectural implications on the performance and cost of graph analytics systems. In *SOCC*, pages 40–51, 2017.

[81] Yanfeng Zhang, Qinxin Gao, Lixin Gao, and Cuirong Wang. Priter: a distributed framework for prioritized iterative computations. In *SOCC*, page 13, 2011.

[82] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. Efficient querying and maintenance of network provenance at internet-scale. In *SIGMOD*, pages 615–626, 2010.

[83] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX ATC*, pages 375–386, 2015.