Datalography: Scaling Datalog Graph Analytics on Graph Processing Systems

Walaa Eldin Moustafa[*], Vicky Papavasileiou[**], Ken Yocum[**] and Alin Deutsch[**]

[*]LinkedIn Corporation
[**]University of California, San Diego

***Abstract*—This paper presents the first Datalog evaluation engine for executing graph analytics over BSP-style graph processing engines. Building on recent advances in Datalog that support efficient evaluation of aggregates functions, it is now easy for data scientists to author many important graph algorithms succinctly. Without the burden of low-level parallelization and optimization, data scientists can avoid programming to the quirks of the latest high-performance distributed computing framework. Where prior approaches build bespoke evaluation engines or modify generalized dataflow processing engines to achieve performance, this work shows how to efficiently evaluate Datalog directly on BSP-style graph processing engines such as Giraph.**

Datalography incorporates both traditional Datalog optimizations, such as semi-naive evaluation, and new evaluation algorithms and optimization techniques for efficient distributed evaluation of Datalog queries on graph processing engines. In particular we develop evaluation techniques that take advantage of *super vertices*, *eager aggregation*, and *asynchronous execution* to optimize graph processing on Pregel-like systems. We implement our algorithms on top of Apache Giraph and our results indicate that Datalography competes with native, tuned implementations, with some analytics running up to 9 times faster.

I. Introduction

Analyzing graph data is a critical capability for today's data scientist. Graph data arises in many contexts, including the day-to-day operation of social media sites, mobile ad networks, financial institutions, health care, genomic, and drug companies, and telecommunication networks. Analyzing and extracting knowledge from graphs require the ability to compute graph queries such as reachability, shortest paths, and measuring connected components, as well as more sophisticated analytics for training machine learning models (recommenders, classifiers, etc.). Many of these algorithms are iterative in nature, repeatedly executing over the graph until reaching a threshold or fixed point.

To that end, recent projects have proposed using Datalog, a recursive language with declarative semantics that allows data scientists to succinctly describe iterative graph analytics [1], [2], [3]. As a recursive language, Datalog naturally expresses graph queries due to the recursive/iterative nature of graph constructs. Moreover, Datalog can express useful aggregate functions that make fixed-point graph analytics easier to author, such as min, max, and $count$. Here we illustrate this expressivity using single-source shortest path, weakly connected components, and PageRank analytics. Each requires up to 4 lines of Datalog while the custom implementations in the Giraph distribution require 50, 60, and 100 lines of Java code respectively. As a declarative language Datalog also enjoys the benefit of allowing the evaluator/compiler to choose the best physical query plan automatically, i.e. without user intervention.

This paper presents the first work to investigate using Datalog, including a useful class of aggregate functions, to author graph analytics for execution on top of unmodified "think-like-a-vertex" parallel graph processing systems. In contrast, existing systems evaluate Datalog programs on custom distributed computing engines, such as a cluster of databases (Myria [3]), custom processing engines (Socialite [2]), or by extending general dataflow systems, such as Spark (BigDatalog [4]) or Hyracks [5] to support recursive evaluation. While we choose Apache Giraph as the vehicle for our investigation because of its maturity (Facebook processes trillion-edge graphs with it [6]), our techniques port straightforwardly to other representatives of this class, such as Pregel [7], GPS [8], GraphLab [9], and GraphX [10].

Our approach aligns the programming abstraction, where Datalog rules operate until they generate no new facts, with the execution model, where Giraph executes until no new messages exist in the system. In addition we explore novel optimizations that exploit the logical computation model and execution capabilities of modern graph processing frameworks. Finally, by building on Apache Giraph we inherit its scaling capabilities and its compatibility with the Hadoop eco-system. This enables Datalography as a step in a data pipeline whose input/output can interface with other Hadoop components (Flume, Kafka, Hive, etc.).

Our results were enabled by the following technical contributions:

1. Datalography incorporates novel program rewriting techniques that enable distributed Datalog evaluation on graph processing systems. The approach transforms each rule of the program into a set of sub-rules the system evaluates locally on each vertex. In addition Datalography applies semi-join and eager aggregation optimizations when possible, effectively creating source-side combiners to reduce communication overhead.

2. Datalography incorporates *super vertices* to amortize the work of rule evaluation. Datalog query compilation includes a step that performs graph *pre-partitioning*. This identifies sets of logical graph vertices that can execute together as a single Giraph physical vertex. Datalography's rewriting (above) enables a collection of individual vertex programs to operate as one super vertex.

3. Datalography leverages Datalog's monotonicity guarantees (Section IV) to provide coordination freeness, allowing asynchronous execution on graph processing systems.

4. We evaluate these concepts on a prototype Datalog evaluation engine built on Giraph Unchained [11]), which incorporates an asynchronous execution mode. The evaluation explores the benefits of our optimizations. We report results for three example queries (on three real-world graphs) and find that Datalography outperforms custom Giraph code for two of them, namely SSSP and WCC, from 1.2x to 9x.

The rest of the paper is organized as follows. Section II reviews Datalog and the properties Datalography leverages. Section III describes how Datalography evaluates Datalog statements and compiles them into logical graph programs. Next, Section IV discusses the optimizations that Datalog with super vertices enables, including eager aggregation, semi-join, source-side combiners, and asynchronous evaluation. We report the results of our Datalog evaluation and execution system in Section V and end with related work.

II. Background

This section gives a brief overview of Datalog and graph processing systems based on the Bulk-Synchronous Parallel computing paradigm (using Giraph as a representative example).

A. Datalog

A Datalog program consists of a set of rules and a set of facts (or tuples). 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 symbol, 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 rule's LHS or head, and $L_1 \ldots, L_n$ are the rule's RHS or body. Note that all the variables in the head must appear in the body.

Each L_i on the rule's RHS is called a subgoal. 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. In database terminology, each predicate symbol corresponds to a relation name. A Datalog program begins with an extensional database (EDB) containing ground facts. Evaluation of a Datalog program creates an intensional database (IDB) containing relations with inferred facts.

Datalog rules can be recursive, i.e., a rule can recursively depend on itself or another rule that depends on it. We illustrate Datalog with the program for Single Source Shortest Path or SSSP. The following program mentions three relations: the EDBs Vertex, Edge, and the IDB SSSP. Predicate $\text{Vertex}(x)$ indicates that x is a vertex identifier. $\text{Edge}(x, y)$ states that a directed edge exists from vertex x to vertex y. We identify the source vertex with vertex id $x = 1$, and we set the distance to that vertex to be 0. Note that the first rule can equivalently be written as SSSP(1, 0).

$$\text{SSSP}(x, 0) \leftarrow \text{Vertex}(x), x = 1.$$
$$\text{SSSP}(x, min\langle d + 1\rangle) \leftarrow \text{Edge}(y, x), \text{SSSP}(y, d).$$

In the above program the EDB relations contain the input or base data while the IDB relation SSSP contains all computed pairs (x, d), where x is a vertex id and d is the minimum distance discovered to x. The non-aggregate arguments in the rule head are used as the grouping variables for aggregation. That is, the system aggregates all d values corresponding to the same x value.

B. Program evaluation

We adopt least fixpoint Datalog semantics and use a bottom-up evaluation strategy. This means that for a given Datalog program P, evaluation continues until a fixpoint is reached – no new facts may be derived for IDB predicates. The evaluation starts with the set of atoms in the EDB, and iteratively adds new atoms using the *immediate consequence operator* T_P, which is defined as follows. Given a database instance, I, an atom A belongs to $T_P(I)$ iff $A \in I$ or there is some rule R in P such that the body of R has a match m against the database and A is the image under m of R's head. Recursive evaluation takes place as shown in Algorithm 1.

Algorithm 1 Recursive Datalog Evaluation

```
$I \leftarrow EDB(P)$
repeat
    $I_{new} \leftarrow T_P(I)$
    $I \leftarrow I_{new}$
until $I_{new} = I$
```

Semi-naive evaluation (SN) is a well-known Datalog evaluation approach [12], where instead of repeatedly evaluating the program using the entire set of atoms found so far, evaluation proceeds by using only newly discovered facts, referred to as the delta. First, SN produces a new delta from $T_P(\Delta I)$ and removes any previously generated facts. It then updates the set of all facts generated, I, and sets the next delta appropriately. This process, seen in Algorithm 2, iterates until it generates no new facts.

Algorithm 2 Recursive Semi-naive Datalog Evaluation

```
$\Delta I \leftarrow EDB(P)$
$I \leftarrow \Delta I$
repeat
    $\Delta I_{new} \leftarrow T_P(\Delta I) - I$
    $I \leftarrow I \cup \Delta I_{new}$
    $\Delta I \leftarrow \Delta I_{new}$
until $\Delta I = \phi$
```

C. Datalog recursion with aggregates

Aggregation is fundamental to many graph algorithms – many compute until an aggregate value, such as a min, max, sum, or $count$, reaches a certain threshold. However, recursive programs with aggregates raise the difficulty that they may not have a least fixpoint solution, or the program evaluation may never terminate. Thus Datalog programs with aggregates has been a topic of research in various studies. Indeed, each of the prior works [1], [4], [3] supports aggregates differently.

The ability to use aggregates in recursive rules is critical to performant Datalog programs for graph analytics. Aggregates are recursive if the head predicate appears in the body. Consider the rewrite of our SSSP algorithm from Section II-A below where we now use a non-recursive min aggregate. While Datalog can evaluate aggregate functions that are non-recursive, this program may not terminate (if the graph is cyclic) and evaluation may be slow since all paths must be evaluated [1].

$$\text{AllPaths}(x, 0) \leftarrow \text{Vertex}(x), x = 1.$$
$$\text{AllPaths}(x, d) \leftarrow \text{Edge}(y, x), \text{AllPaths}(y, d_1), d = d_1 + 1.$$
$$\text{SSSP}(x, min\langle d\rangle) \leftarrow \text{AllPaths}(x, d).$$

Here Datalography stands on the shoulders of prior work. In particular it supports the *meet* aggregate operations (idempotent, commutative, and associative) presented in Socialite [1] (min,max), as well as the non-meet operations such as sum and $count$ aggregates through explicit local stratification [13], which we illustrate with PageRank in Section III. Note, Datalography can support the additional aggregates without the use of imperative code or memoization of intermediate values (see comparison in Section VI).

D. Giraph

Datalography is the study of Datalog graph analytics on BSP-style graph processing systems, such as Apache Giraph. We chose Giraph as it is a well-supported, efficient, open-source implementation of *think-like-a-vertex* graph processing [7]. Distributed graph processing frameworks are especially suitable for parallelizing graph algorithms such as PageRank or Shortest-Paths. Similarly, numerous works have shown that these frameworks can also express complex machine learning and data mining algorithms.

In this model of distributed graph processing, data resides on edges and vertices, and each vertex repeatedly executes the same program. Thus each vertex stores the result of its evaluation locally. A vertex program executes in three phases: i) receive messages from neighbor vertices, ii) compute and possibly produce new values for the vertex and outgoing edges, and iii) create messages for neighbor vertices. A Giraph program terminates when a user-specified termination condition occurs or there are no new messages to deliver.

Thus Giraph programs are imperative, and evaluation follows the bulk-synchronous processing (BSP) model in which the computation is broken into super-steps (iterations) interleaved with global synchronization barriers. The system partitions the input graph (by vertex) across all workers, each of whom processes their subset of vertices. A master process orchestrates the super-steps, maintains global aggregations, and computes terminating conditions. A configurable number of worker processes execute the vertex program and communicate intermediate results, in the form of messages, with each other.

In addition to the standard Giraph distribution, we leverage Giraph Unchained [11], which allows asynchronous execution of vertex programs. This allows a vertex to read and process messages as soon as they arrive without having to wait for the next super-step.

III. Distributed Datalog Evaluation on Graph Processing Systems

A. Data Distribution

We partition data across a set of *super vertices*, $\mathbb{V}$. We distinguish between graph data vertices, $\mathbb{X}$, which we refer to as *vertices*, and super vertices, an abstraction that contains data from multiple vertices. Thus each super vertex contains a horizontal partition of the EDB and IDB data relevant to its vertices. For a vertex id $x \in \mathbb{X}$, $V(x)$ denotes the super vertex to which it belongs.

Our model partitions the data based on the presence of vertex id terms in predicates. These predicates fall into two classes. One class of predicates, *vertex-class* predicates, logically represent vertex data by including a single vertex id term. We partition vertex-class predicates by that vertex id. Specific examples include predicates such as $Vertex$, $Person$, $Movie$, or $PageRank$ predicates.

The second class of predicates, *edge-class* predicates, logically represent edge data and contain source and destination vertex id terms. Such edge-class predicates represent relationships, e.g., $Edge$, $Follows$, or $Likes$. Datalography stores such a predicate data as two partitioned EDBs. For example, in the case of an $Edge$ predicate, the first is partitioned on the source vertex id resulting in an *InEdge* EDB/predicate, and the second is partitioned on the destination vertex id resulting in an *OutEdge* EDB/predicate.

Note that it is also necessary for each super vertex to be able to find the super vertex that contains its vertices' neighbors (both incoming and outgoing). To do so, we keep a map $M_{xv} : \mathbb{X} \rightarrow \mathbb{V}$ in each super vertex v, which maps every neighbor vertex id y of a vertex with id x in a super vertex v to the super vertex that contains y, $V(y)$.

For IDB predicate partitioning, Datalography augments Datalog rules with a location specifier – a special symbol preceding an argument in the rule head that specifies how to partition the rule result. For example, $P(\#x, y) \leftarrow Edge(x, z), Edge(z, y)$ denotes that the system will partition the rule result by the values x_i of variable x, storing them at super vertex $V(x_i)$.

B. Architecture

Our distributed Datalog evaluation architecture resembles Giraph's. A master process maintains global aggregations and determines termination, while a number of worker processes evaluate the programs rules and communicate intermediate results to each other.

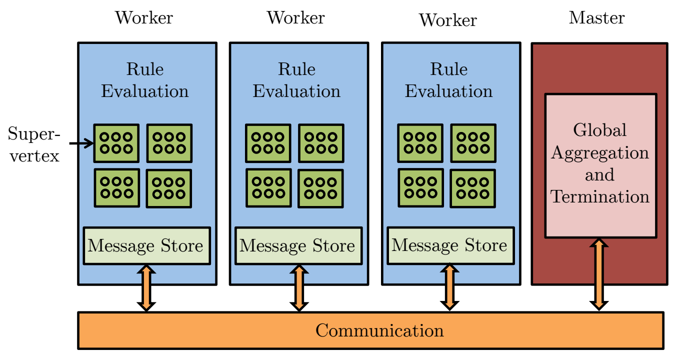


Fig. 1. Datalography Distributed Architecture

Notation	Meaning
$\mathbb{V}$	Set of super vertices
$\mathbb{X}$	Set of vertex ids
v_i	Super vertex
$x, x_i, x_{i_l}, x_{j_l}, x_{j_k}$	Variables representing vertex ids
z_i	Vertex id
P_i	Vertex-class predicate
E	Edge-class predicate
G_D	Data Graph
G_R	Query Graph
D_R	DAG of Query Graph

TABLE I
Notations used in Section III-C1

Each worker process uses multiple compute threads to process super vertices in parallel. Our super-vertex abstraction maps to Giraph's vertex abstraction – a single Giraph vertex consists of multiple data graph vertices (their EDB/IDB data). The information contained in one Giraph vertex is no longer only about one data vertex, but a group of vertices (a super vertex). This allows for batch processing, amortizing the cost of Giraph vertex computation over multiple data vertices at a time. A separate module handles communication with its own thread pool (Netty threads). Each worker has its own messaging store, which it uses to receive incoming messages from other workers. Figure 1 shows this high-level architecture.

Our approach for distributed Datalog evaluation is based on rewriting the program to a set of rules that can each be evaluated locally by each worker. Workers exchange information that is necessary to evaluate non-local rules (i.e., rules which require information that is not stored on the same worker). We describe the approach in detail in the following two subsections.

C. Datalog Evaluation

First, we discuss how Datalography evaluates individual rules, and then we discuss how the system evaluates entire programs. For reference, Table I lists the notations used throughout this section.

*1) **Individual Rule Evaluation**:* The goal of rule evaluation is the parallel, independent execution of a rule by each super vertex. The technical challenge is that the data needed to evaluate a rule R at a super vertex v may not be co-located at v. Rather than use an inefficient brute-force broadcast of data across super vertices, our solution interleaves parallel rule execution stages with communication stages. Each execution stage has two goals. First, advance the computation of the rule while filter intermediate results to minimize the data transmitted in the second stage. Second, communicate intermediate data by identifing recipient super vertices relevant to subsequent execution stages. This replaces the aforementioned broadcasting with targeted sends..

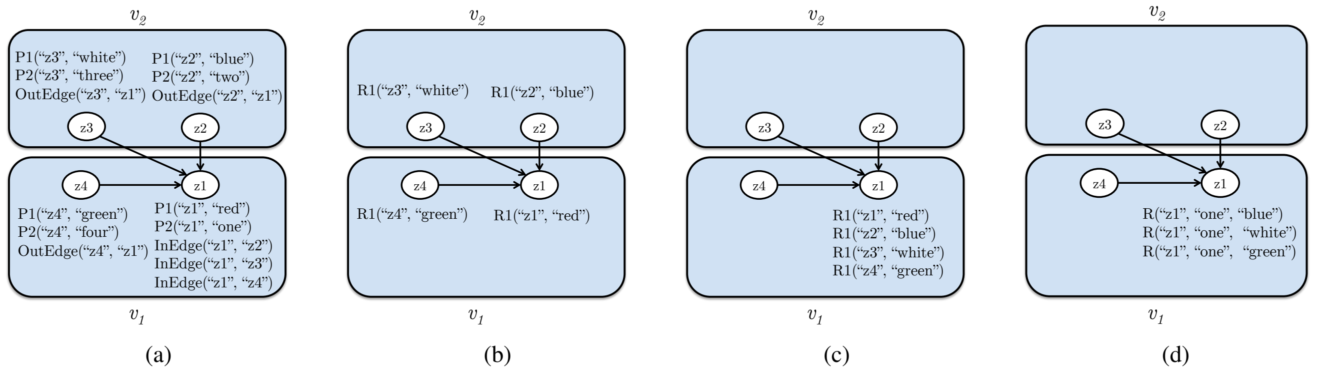


Fig. 2. Rule evaluation steps for rule $R(\#x_2, a, b) \leftarrow P_1(x_1, a), Edge(x_1, x_2), P_2(x_2, b)$.

A feature of our solution is that it expresses each execution phase as a Datalog rule itself. More precisely, Datalography automatically compiles the original rule R into a set of simpler rules that can be evaluated in order. This allows each super vertex to evaluate the first rule independently (without prior data communication). Then the system can push data between super vertices that is necessary to complete the evaluation of subsequent rules at each super vertex.

We introduce our approach using an example, and then describe the general algorithm.

Example 1. Consider the evaluation scenario in Figure 2. The data graph consists of 4 vertices, whose ids are z_1 to z_4. The EDB consists of three predicates, P_1, P_2, and $Edge$. Both P_1 and P_2 are partitioned by the vertex id (i.e., z_i), while $Edge$ is partitioned by the source vertex id resulting in $OutEdge$, and by the destination vertex id resulting in $InEdge$ (all $InEdge$ atoms are with z_1 since it is the only vertex with incoming edges). Super vertex v_1 contains partitions of z_1 and z_4, while super vertex v_2 contains partitions of z_2 and z_3. This arrangement is shown in Figure 2(a). Notice that rule $R(\#x_2, a, b) \leftarrow P_1(x_1, a), Edge(x_1, x_2), P_2(x_2, b)$ cannot be evaluated locally at each super vertex, since the data it needs is distributed across super vertices (e.g., the rule should derive, among others, the fact $R(\text{“}z_1\text{”}, \text{“}blue\text{”}, \text{“}one\text{”})$, which requires facts $P_2(\text{“}z_1\text{”}, \text{“}one\text{”})$ from v_1 and $P_1(\text{“}z_2\text{”}, \text{“}blue\text{”})$ from v_2).

In our solution, the rule is automatically rewritten to:

(i) $R_1(\#x_1, a) \leftarrow P_1(x_1, a)$ and

(ii) $R(\#x_2, a, b) \leftarrow R_1(x_1, a), InEdge(x_2, x_1), P_2(x_2, b)$,

where rule (i) will be evaluated first. As we can see, rule (i) can be evaluated independently by each super vertex since the data it needs is available locally. The results of evaluating this rule are shown in Figure 2(b). Since the next rule to evaluate depends on $R_1(x_1, a)$ that is partitioned by x_1, while the result R needs to reside at $V(x_2)$, evaluation results of R_1 (partitioned by x_1) are pushed along the *outgoing* edges, so that they can be consumed at super vertices containing information about x_2. The resulting communication is shown in Figure 2(c). After this communication, rule (ii) can be evaluated, since $P_2(x_2, b)$ is already partitioned by x_2, $InEdge(x_2, x_1)$ is already partitioned by x_2, and finally, for each x_1, $R_1(x_1, a)$ is available at $V(x_2)$ since it was communicated from each x_1 to the super vertex containing its outgoing neighbor x_2. The result of evaluating R is shown in Figure 2(d). □

To describe the algorithm, we introduce some notation first. A rule body consists of a set of EDB or IDB vertex-class predicates P_1 to P_n, which are partitioned based on the location of x_{i_1} to x_{i_n}, respectively, and those predicates are related by the edge-class $Edge$ predicate (E for short) as follows:

$$\begin{aligned} R(\#x, \overline{x}) \leftarrow & P_1(x_{i_1}, \overline{x_{i_1}}), \ldots, P_n(x_{i_n}, \overline{x_{i_n}}), \\ & E(x_{j_1}, x_{k_1}, \overline{x_{jk_1}}), \ldots, E(x_{j_m}, x_{k_m}, \overline{x_{jk_m}}) \end{aligned}$$

where $i_1 \ldots i_n, j_1 \ldots j_m, k_1 \ldots k_m \in \{1 \ldots n\}$, P_l is partitioned by x_{i_l}, and $\overline{x_{i_l}}$ are the remaining arguments for P_l. R is to be partitioned by x, and $\overline{x}$ are the remaining arguments of R. We note that x denotes one of $x_{i_1} \ldots x_{i_n}$, but we use x to distinguish it as the one determining the location of the results. We assume that the $P_1 \ldots P_n$, and E predicates represent a data graph G_D.

We observe that the rule body defines a *query graph* (or rule graph), G_R, whose nodes correspond to the partition variables of the vertex-class predicates, and edges correspond to the $Edge$ predicates appearing in the rule:[1] the nodes are $\{x_{i_l} | 1 \leq i_l \leq n\}$, and edges are $\{(x_{j_l}, x_{k_l}) | 1 \leq j_l, k_l \leq n\}$. For example, the rule from Figure 2, has a query graph with two nodes x_1 and x_2, with and edge from x_1 to x_2.

The idea of the rewriting algorithm is to produce a sequence of rules $\langle R_1, \ldots, R_n \rangle$, each corresponding to a node in the query graph, where R_1 can be evaluated independently without any data communication, and for all subsequent rules R_i, where $1 < i \leq n$, R_i depends only a subset of $\{R_j | j < i\}$. Thus all rules on which R_i depends will have executed and sent their relevant results to the appropriate super vertices prior to R_i's execution – this enables R_i to execute locally. Say we determine rule R_i is partitioned by x_i. Then, at evaluation time, $V(x_i)$ stores the results of rule R_i. Algorithm 3 shows the algorithm for rewriting a rule and works as follows:

1) The algorithm constructs the rule graph G_R from the rule body predicates (line 1).
2) It then removes the graph G_R edge directions and converts the resulting graph to a DAG (by re-assigning edge directions based on BFS levels), D_R, with the node corresponding to x being the sink node (line 2). D_R contains precisely the vertices of G_R and a cycle-free version of its edges.
3) The algorithm creates a topological order for the DAG (line 3), and generates a rule for each node in that topological order (lines 4-18). A rule R_i is partitioned by x_i and can only contain in its RHS the predicates P_l partitioned by x_i, or Edge predicates between x_i and x_j, where x_j is a predecessor to x_i in G_R, in addition to all R_j predicates corresponding to x_j.

As we stated earlier, for every rule R_i that depends on R_j, R_j's results at $V(x_j)$ should be sent to $V(x_i)$. Note that the dependency of x_i on x_j means that there is an edge (x_j, x_i) in D_R; that means that either (x_i, x_j) or (x_j, x_i) exists in G_R:

[1]We use "nodes" to refer to the query graph vertices to avoid confusion with the data graph vertices.

Algorithm 3 Rule rewriting algorithm

```
1: $G_R \leftarrow Graph(R)$
2: $D_R \leftarrow DAG(Undirected(G_R), x)$
3: $[x_1 \dots x_n] \leftarrow topsort(D_R))$
4: for $1 \le i \le n$ do
5:     $rhs \leftarrow P_i(x_i, \dots)$
6:     for $x_j \in predecessors(x_i, D_R)$ do
7:         $rhs \leftarrow rhs \cup R_j(x_j, \dots)$
8:         if $(x_i, x_j) \in G_R$ then
9:             $rhs \leftarrow rhs \cup InEdge(x_j, x_i)$
10:        else if $(x_j, x_i) \in G_R$ then
11:            $rhs \leftarrow rhs \cup OutEdge(x_j, x_i)$
12:        end if
13:    end for
14:    Generate rule: $R_i(\#x_i, vars(rhs)) \leftarrow rhs$
15: end for
16: if $R$ is aggregate rule then
17:    Apply aggregate on $R_n$
18: end if
```

1) In case $(x_i, x_j) \in G_R$, then $E(x_i, x_j)$ is rewritten as $OutEdge(x_i, x_j)$ and results from $V(x_j)$ are sent to $V(x_i)$ by using x_j's *incoming* edges.
2) On the other hand, if $(x_j, x_i) \in G_R$, then $E(x_j, x_i)$ is rewritten as $InEdge(x_i, x_j)$ and results from x_j are sent to x_i by using x_j's *outgoing* edges.
3) In both cases, the map M_{xv} is used to lookup $V(x_i)$, so that the destination super vertex can be determined.

In the rest of the paper, unless otherwise denoted, the location specifier is the first argument of the head and thus omitted.

Example 2. PageRank ranks web pages according to their popularity: the more incoming links a webpage has, the more popular it is. At iteration 0, all vertices initialize their rank with 1. For every subsequent iteration, a vertex reads the rank of its incoming neighbors (computed in the previous iteration) and updates its current rank. The PageRank algorithm can be concisely expressed by the following Datalog program (versus 100 lines of Java code used in the Giraph distribution):

$$\begin{aligned}
&\text{Outdegree}(x, \mathit{count}\langle y\rangle) \leftarrow \text{Edge}(x, y).\\
&\text{PageRank}(x, 0, 1).\\
&\text{PageRank}(x, i+1, \mathit{sum}\langle p/d\rangle) \leftarrow \text{Edge}(y, x),\\
&\qquad\qquad \text{PageRank}(y, i, p), \text{Outdegree}(y, d), i < 20.
\end{aligned}$$

Algorithm 3 replaces the final rule with the last two rules below (which we named mnemonically for the sake of readability):

$$\begin{aligned}
&\text{Outdegree}(x, \mathit{count}\langle y\rangle) \leftarrow \text{OutEdge}(x, y).\\
&\text{PageRank}(x, 0, 1).\\
&\text{PageRank\_Y}(y, i, p, d) \leftarrow \text{PageRank}(y, i, p),\\
&\qquad\qquad \text{Outdegree}(y, d), i < 20.\\
&\text{PageRank}(x, i+1, \mathit{sum}\langle p/d\rangle) \leftarrow \text{InEdge}(x, y),\\
&\qquad\qquad \text{PageRank\_Y}(y, i, p, d).
\end{aligned}$$

The $PageRank\_Y$ rule can be executed locally, for each y at its host super vertex. It sends its results, partitioned by the values y_i of y, to the super vertices hosting the neighbors of the y_i's. This sets up the final rule to execute locally at the next iteration. □

Each sub-rule generated by Algorithm 3 can be evaluated locally at every super vertex. Evaluation proceeds in the standard way by compiling rules to logical query plans based on relational algebra operators [14] (predicate arguments bound to the same variable are translated to equi-joins, conditional predicates are translated to selections (e.g., $x \neq y$ or $x < a$), and LHS arguments are translated to projections, aggregations when applicable, and materialization operators). Standard algebraic optimizations are applied (e.g. pushing selections/projections before joins). A simple cost-based optimizer selects the best physical execution plan.

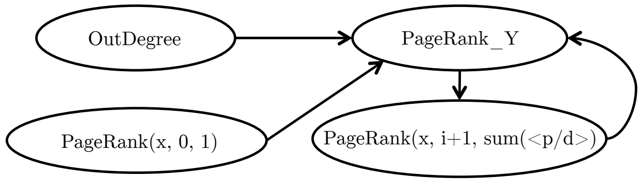


Fig. 3. PageRank Rule Dependency Graph (rules are identified by their head)

*2) **Program Evaluation:*** A program is evaluated by rewriting each rule in the program using Algorithm 3. The result of the rule rewriting of a program P is another rewritten program P' whose rules can be evaluated in parallel at each super vertex, and whose partial results can be communicated along the edges for further rule evaluation.

To schedule rule execution, we determine the data flow between rules. A *rule dependency graph* is constructed out of the rewritten program P', where nodes represent rules, and an edge is created between two nodes (R_1, R_2) if rule R_2 depends on R_1 (i.e., head predicate of R_1 appears in R_2 rule body). See Figure 3 for the dependency graph of the rewritten PageRank from Example 2.

Recursion is reflected in the rule dependency graph as a strongly connected component (i.e., there is a path from any node in the component to all other nodes in the component). By collapsing each strongly connected component into an individual node (and maintaining edges across components), the resulting graph is guaranteed to be a DAG. Program evaluation proceeds from the DAG source node(s), evaluating a component when all its predecessors are evaluated. If a component is a strongly connected component (vs. a singleton rule), the component is evaluated via fixed point evaluation till it converges (i.e. no new facts are derived). To determine whether a component has converged, all super vertices inform the master whether they have derived new facts. If at least one super vertex has derived new facts, then fixed point evaluation continues. Otherwise, evaluation of the respective component concludes and the engine proceeds to further components in the DAG.

Example 3. The dependency graph for PageRank from Figure 3 yields the following rule evaluation discipline. Its strongly connected components are $C_1 = \{Outdegree\}$, $C_2 = \{PageRank(x, 0, 1)\}$, $C_3 = \{PageRank\_Y, PageRank(x, i+1, sum < p/d >)\}$ (we refer to rules by their head, dropping the variables when there is no ambiguity). The dependencies between them are $C_1 \rightarrow C_3$, $C_2 \rightarrow C_3$, yielding a topological sort in which C_1 and C_2 are evaluated first (as single rules, locally). Next, the two rules in C_3 are evaluated to a fixpoint.

To fit this evaluation technique within Giraph's framework, the program is rewritten at the worker level. A separate worker-level module determines which rules to evaluate during the next super-step using the DAG topological order, whether a recursive component (i.e., strongly connected component) is being currently evaluated, and whether the component has reached a fixed point. To minimize super-step synchronization overhead it processes the maximum possible number of rules during every super-step. Therefore, co-located rules are automatically evaluated during the same super-step, while non-co-located rules communicate their results to their dependencies by exchanging data across the graph edges to be consumed in subsequent super-steps.

IV. Optimizations

This section discusses four 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. Eager Aggregation

Eager aggregation was first addressed by [15] 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, obtaining a significant reduction in the volume of data transmitted between worker nodes. Datalography does so by applying aggregation before data transmission and then another aggregation is required to aggregate the partially aggregated data from the senders.

Observe that Algorithm 3 relegates aggregation to the final rule, R_n. We rewrite the Datalog program to perform aggregation even in the intermediate rules R_i (where $i < n$). The aggregated results sent by intermediate rules can be significantly smaller than their unaggregated counterpart.

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_a 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. Therefore, given a rule R with a decomposable aggregate, and which is rewritten by Algorithm 3 to rules $R_1 \dots R_n$, we adapt techniques from [15] to push down the aggregate from R_n to other rules R_i, where $i < n$.

Example 4. Recall from Example 2 the PageRank version obtained after rewriting according to Algorithm 3. Notice that the PageRank_Y rule performs no aggregation, and recall that its execution sends multiple tuples with the same value y_i for y to the super vertices hosting neighbors of y_i. Say x_0 is the neighbor of y_1 and y_2, then super vertex $V(x_0)$ receives a bag of tuples S_1 for y_1, and a bag of tuples S_2 for y_2, aggregating $S_1 \uplus S_2$.

In contrast, eager aggregation produces the program below.

$$\begin{array}{l}
\text{Outdegree}(x, \mathit{count}\langle y\rangle) \leftarrow \text{OutEdge}(x, y).\\
\text{PageRank}(x, 0, 1).\\
\text{PageRank\_Y}(y, i, \mathit{sum}\langle p/d\rangle) \leftarrow \text{PageRank}(y, i, p),\\
\qquad\qquad\qquad\qquad\qquad\qquad \text{Outdegree}(y, d), i < 20.\\
\text{PageRank}(x, i+1, \mathit{sum}\langle p\rangle) \leftarrow \text{InEdge}(y, x),\\
\qquad\qquad\qquad\qquad\qquad\qquad \text{PageRank\_Y}(y, i, p).
\end{array}$$

Notice that, before sending, the execution of the PageRank_Y rule at $V(y_i)$ aggregates S_i into a single tuple each. Now $V(x_0)$ receives only two tuples, which are summed up. The semantics of the program is preserved since $\Sigma_{p\in S_1 \uplus S_2} \frac{p}{d} = (\Sigma_{p\in S_1} \frac{p}{d}) + (\Sigma_{p\in S_2} \frac{p}{d})$. □

B. Semi-join

In Section III-C1, we discussed how Datalography executes local rules by sending intermediate evaluation results to the correct super-vertex partitions based on the vertex id. Given a rule R_i that is a predecessor of rule R_{i+i} and that is evaluated for a vertex v, the original approach decides which neighbors w should receive results of R_i's evaluation. Datalography achieves this by joining the results of R_i at $V(v)$ with $V(v)$'s Edge relation (InEdge or OutEdge, as dictated by Algorithm 3). Upon receipt at $V(w)$, the sent tuples undergo a second join, this time with the appropriate Edge relation stored at $V(w)$ (if $V(v)$ sent data along the out-edges, R_{i+1} joins the received data with the InEdge relation, and vice versa). It is easy to see that when restricted to v-relevant tuples, the join at the sender already computes the same tuples as the w-relevant restriction of the join at the receiver.

We avoid the redundant join using a technique reminiscent of the classical semi-join reducer technique from relational distributed database processing [16]. The optimization can once again be expressed by rewriting the original program. The rewrite moves the edge predicate from the destination rule R_{i+1} to the source rule R_i (flipping In- and Out- directions). The values of the destination vertex are added to the result of R_i so they can be directly retrieved by R_{i+1} without having to perform a local join.

Example 5. The semi-join optimization applied to the PageRank version from Example 2 yields

$$\begin{array}{l}
\text{Outdegree}(x, \mathit{count}\langle y\rangle) \leftarrow \text{OutEdge}(x, y).\\
\text{PageRank}(x, 0, 1).\\
\text{PageRank\_Y}(y, x, i, p, d) \leftarrow \text{OutEdge}(y, x),\\
\qquad\qquad\qquad\qquad\qquad \text{PageRank}(y, i, p),\\
\qquad\qquad\qquad\qquad\qquad \text{Outdegree}(y, d), i < 20.\\
\text{PageRank}(x, i+1, \mathit{sum}\langle p/d\rangle) \leftarrow \text{PageRank\_Y}(y, x, i, p, d).
\end{array}$$

Notice that the join with InEdge from the last rule was transformed into a join with OutEdge in the PageRank_Y rule, which now also outputs the x vertices. The last rule can now read the y, x pairs directly from the results sent by the PageRank_Y rule, without having to compute them via join. □

C. Source-side Combiners

Eager aggregation and semi-join optimization are beneficial in isolation from each other, but can significantly reduce communication cost when enabled simultaneously. Section V shows the effect each optimization has on runtime.

Their combined effect yields an optimization we call *source-side combiners*. Source-side combiners ensure that the information destined to a super vertex is partially aggregated at the sender side before sent to the destination super vertex. This technique is named after *map-side combiners* in MapReduce programs [17]. However, MapReduce, directly exposes such combiners to the developer in a non-declarative fashion. In contrast, Datalography automatically discovers source-side combiners.

Example 6. Applying both eager aggregation and semi-join optimization to the PageRank version of Example 2 yields

$$\begin{array}{l}
\text{Outdegree}(x, \mathit{count}\langle y\rangle) \leftarrow \mathit{OutEdge}(x, y).\\
\text{PageRank}(x, 0, 1).\\
\text{PageRank\_Y}(x, i, \mathit{sum}\langle p/d\rangle) \leftarrow \text{OutEdge}(y, x),\\
\qquad\qquad\qquad\qquad\qquad\qquad \text{PageRank}(y, i, p),\\
\qquad\qquad\qquad\qquad\qquad\qquad \text{Outdegree}(y, d), i < 20.\\
\text{PageRank}(x, i+1, \mathit{sum}\langle p\rangle) \leftarrow \text{PageRank\_Y}(x, i, p, d).
\end{array}$$

□

D. Asynchronous execution

Giraph is based on synchronous execution in which the master coordinates evaluation through the successive execution of super-steps. Between super-steps, a barrier blocks all computation of vertex programs and allows only the master to compute. This approach simplifies synchronization of vertex execution and communication. Vertices communicate with each other using messages – a message sent during one super-step can be accessed only at the next super-step. Thus although reading, computing, and writing of vertex data happens in every super-step, a vertex can only read a value produced in a previous super-step. This ensures that race conditions between computing vertices do not occur.

Although synchronous execution simplifies the system's design, it also suffers from costly synchronization overheads. First, stragglers can occur as slow running workers cause finished works to go idle waiting for them to finish. This becomes especially prevalent for graphs that follow a powerlaw distribution where few vertices have a high degree of edges leading to unequal worker load. Second, it is difficult to pipeline computation and communication when each super-step requires a global synchronization barrier.

These shortcomings are common to BSP-style graph processing systems and there have been various approaches to add asynchrony to vertex processing. Here we take advantage of a particular approach, the Barrierless Asynchronous Parallel (BAP) model, implemented in Giraph Unchained [11]. This work extends Giraph to support asynchronous execution by allowing messages to be read as soon as they arrive and by reducing the number of global super-step barriers. This means that where previously vertices were computing once during a super-step, they now compute multiple times until no more new data can be produced (all messages have been read).

The CALM-conjecture [18] states that monotonic programs allow for asynchronous execution since they are invariant to message ordering and retry and hence are *eventually consistent*. We exploit this fact by using asynchronous evaluation of our recursive monotonic Datalog programs.

V. Experiments

While the conciseness and maintainability benefits of expressing graph algorithms in Datalog are indisputable, a natural question is to quantify the performance penalty (if any) of these benefits.

Graph processing systems such as Giraph are highly tuned for the efficient evaluation of graph algorithms. Giraph is a mature system used by companies, such as Facebook, in real-world scenarios. It has undergone performance optimizations that, among other things, optimized the memory footprint of the graph representation and distributed communication. For a detailed description of optimizations refer to [6].

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 can be more efficient than running their imperative implementations directly in Giraph. Moreover, it shows that Datalog evaluation engines benefit when exploiting optimizations enabled by the declarative nature of Datalog and its monotonicity (i.e., super vertices, eager-aggregation, asynchronous execution).

A. Experimental Setup

All experiments were run on a cluster of 8 nodes, each with 32GBB 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 Unchained based on Giraph 1.1. The changes made by Giraph Unchained do not apply to the synchronous execution mode, hence the code is similar to that of Giraph 1.1.

We used real-world datasets[2], stored as text files in HDFS, namely hollywood-2009 (HW-9), hollywood-2011 (HW-11) and arabic (AR). Their characteristics can be seen in Table II. `HW-9` and `HW-11` are social graphs with large average degree and small diameter. `AR` on the other hand has a large diameter with small degree since it represents web sites that could contain pages written in Arabic.

TABLE II
Dataset characteristics

Dataset	V	E	Avg Degree	Avg Diameter
HW-9	1.1M	113.9M	100	3.87
HW-11	2.2M	229M	105	3.92
AR	22.7M	640M	28.14	16.58

For the assignment of vertices to super vertices we used the ParMetis [19] graph partitioning library. Section III made the case for super vertices as they enable a more sophisticated partitioning of the vertices: every super vertex contains vertices of the initial input graph that are "close" to each other to minimize communication between different super vertices.

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 versus executing the Datalog implementation on Datalography with synchronous execution and on Datalography with asynchronous execution.

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 was shown in Section II-A.

We compared against the implementation provided by Giraph in its examples package. The source from which the algorithm starts is the vertex with ID 1 and 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.

Weakly Connected Components finds components in a graph, that is sub-graphs 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.

The Datalog query that evaluates the WCC algorithm is:

$$\begin{array}{l}\mathrm{WCC}(x,x) \leftarrow \mathrm{Vertex}(x).\\ \mathrm{WCC}(x, min\langle l\rangle)) \leftarrow \mathrm{Edge}(y,x), \mathrm{Wcc}(y,l).\end{array}$$

The first rule initializes the component of a vertex to its id. The second rule recursively traverses the graph and assigns to each vertex the component that is the minimum of its incoming neighbors. The

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

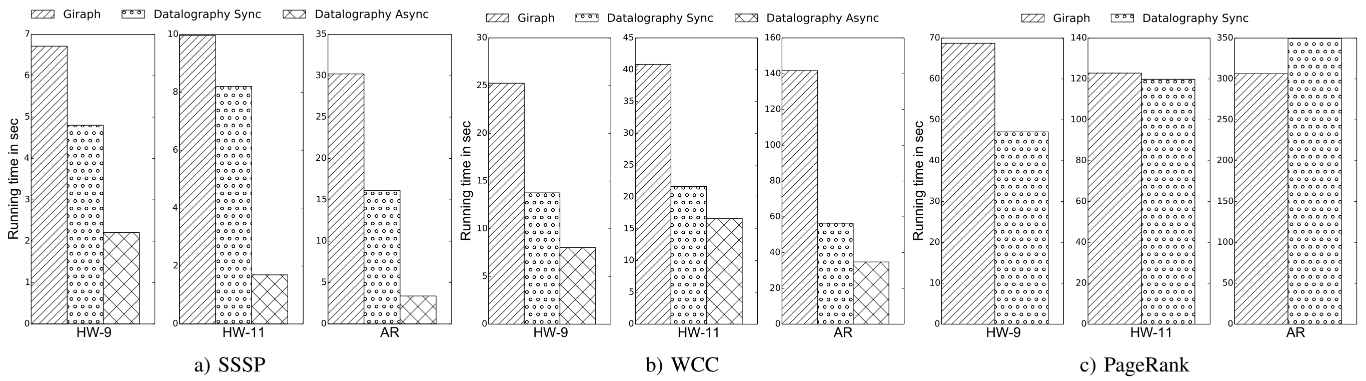


Fig. 4. Running time comparison of Giraph, Datalography Sync, and Datalography Async

evaluation stops when the component of all vertices don't change anymore.

PageRank is described in Section III-C1, where we show its Datalog implementation. We compare it against the customized implementation provided in the Giraph example package.

B. Comparison to Giraph

Super vertices in Datalography correspond to a vertex in Giraph. The vertex data type in Datalography is a database that contains a set of tables, namely vertices, edges, predicates and messages. The vertices and edges tables are instantiated when the input graph is loaded. The predicates table corresponds to predicates in the Datalog query and gets populated in every superstep of the evaluation. The messages table contains the information about the messages that are sent at the end of every superstep such as the destination ID and the value of the message. The vertices and edges are constant during evaluation and the messages table get created anew at every superstep. However, the predicates tables keep increasing for every iteration as new vertices get woken up and compute their values. Memory utilization is arguably an issue in Datalography and has not been optimized as well as Giraph's. The tables store information in human-readable format whereas Giraph employs ByteArrays. Moreover, Giraph avoids the creation of new objects for messages, a benefit Datalography does not take advantage of. Despite the fact that our implementation does not (yet) exploit these optimization opportunities, Datalography outperforms Giraph for all algorithms and datasets (except near break-even performance with pagerank).

For all datasets and algorithms, we configured Datalography to use 1000 super vertices. This offered the near best performance across all combinations of dataset and algorithm (see Section V-C). This decision does not affect the performance of Giraph as it is agnostic to the number of super vertices.

SSSP: Datalography outperforms Giraph for all datasets (Figure 4a). The largest gains in performance are observed with `AR` that has the smallest degree. Datalography outperforms Giraph by a factor of 1.9x with synchronous mode and a factor of 9x when the asynchronous mode of execution is used. With `HW-9`, `HW-11` Datalography is 1.2x faster in synchronous mode and 3x and 5.8x in asynchronous mode respectively. SSSP needs 15 supersteps for `HW-9`, 19 supersteps for `HW-11` and 39 for `AR`.

WCC: Again, Datalography outperforms Giraph on all datasets (Figure 4b). Performance follows the same trend as with SSSP where larger gains are observed with smaller degrees of a graph. Datalography is faster by a factor of 2.5x in synchronous mode and up to 4x faster in asynchronous mode for `AR`.

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, since it computes new PageRank values for all nodes at every iteration. Moreover, since recursive programs with sum aggregate are not monotonic in general, the $\mathrm{PageRank}$ predicate uses an iteration variable i to compute the PagerRank value in stages. This renders asynchronous execution unsuitable since the iteration variable already creates a logical synchronization barrier. Despite these difficulties, Datalography performs on par with Giraph for `HW-11`, outperforms it by a factor of 1.4x for `HW-9` and looses by a factor of 0.9x for `AR`.

C. Scaling super vertices

In these experiments (summarized in Figure 6), we scale the number of super vertices to show the effect they have on runtime for both synchronous and asynchronous mode. The measurements show that a number of super vertices between 800 and 1600 gives the best performance. This makes sense as too few super vertices do not take advantage of parallelism whereas too many super vertices hinder the computation and communication optimizations otherwise offered.

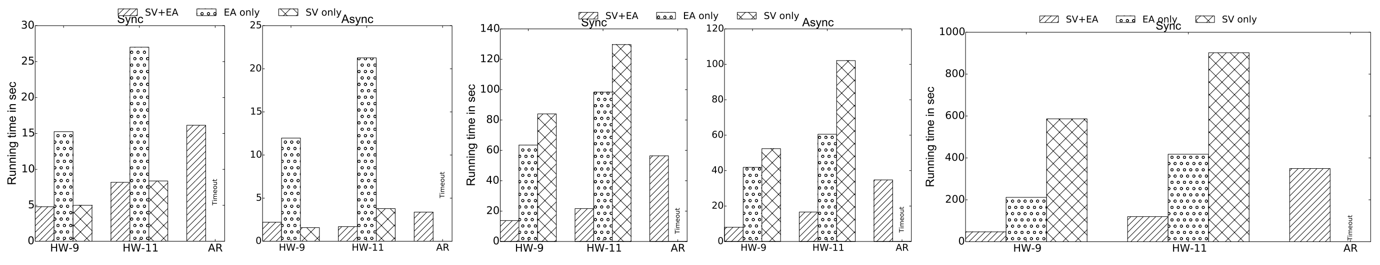
D. Effect of optimizations on runtime

We compare the running time of SSSP, WCC and PageRank when the evaluation is not using super vertices (EA only) or not using eager-aggregation (SV only) to measure the effect of each optimizations. Figure 5 summarizes the results. When not using super vertices, the input graphs have the same vertices as Giraph, outlined in Table II.

On dataset `AR`, no run terminated after 20mins of execution for all cases except for SSSP and EA-only. This highlights the importance and impact the optimizations have on the running time and scalability of Datalography. Neither of the optimizations suffices in isolation, but their synergistic combination (corresponding to source-side combiners) is the key to good performance. This holds for the datasets `HW-9` and `HW-11` as well where combining the optimizations is critical.

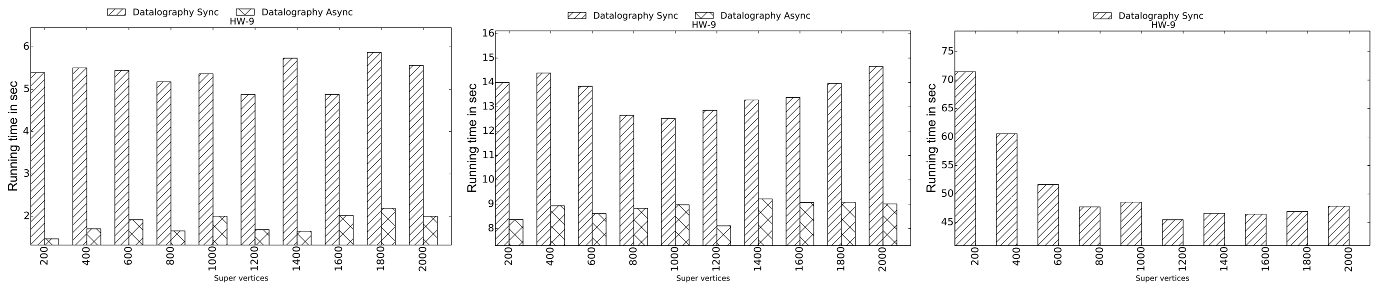
VI. Related Work

Datalog evaluation frameworks. Existing distributed Datalog frameworks tend to use bespoke engines, and only one of which appears to run over think-like-a-vertex graph processing platforms [20]. That system focused on generic datalog programs, not graph analytics – it had no support for recursive aggregates, rule rewriting and local



a) SSSP b) WCC c) PageRank

Fig. 5. Effect of optimizations on runtime



a) SSSP b) WCC c) PageRank

Fig. 6. How the number of super vertices affects runtime

rule evaluation, or the many optimizations Datalography uses to efficiently process such analytics. Interestingly, other works explored the opposite direction, transforming existing think-like-a-vertex programs into datalog analytics that then execute as more traditional relational dataflows on systems like Hyracks [5]. Below we discuss the datalog frameworks that can support graph analytics but run on either bespoke or modified data processing engines.

Socialite [1] and Myria [3] are specialized engines catered to the 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 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 them for a subset of syntactically-restricted programs by stratifying the Datalog query using the *time* construct. In contrast, Datalography allows for recursive sum and $count$ aggregates for queries that are explicitly locally stratified [13] without the use of imperative code.

BigDatalog [4] is built on top of Spark [21], 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 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 [22] for recursive aggregates by combining monotonic recursive versions, i.e., $mmin$, $mmax$, $mcount$, $msum$, with a final non-recursive aggregate. The evaluation memoizes intermediate aggregated values hence, a final rule that applies a non-recursive aggregate is needed to return the correct result. Datalography does not suffer from this inefficiency as the aggregated values computed by every iteration override the previous values. Furthermore, Datalography can support $mcount$ and $msum$-style aggregates through a combination of max and stratified sum aggregates.

Datalog languages. Recently, Datalog has been the language of choice for approaches that offer programmers a declarative and concise abstraction over the impeding mismatch involved in building large, commercial scale products. Yedalog [23] and Bloom [24] use Datalog to express data analysis and communication in a single platform allowing programmers to focus on the high-level logic instead of details of distribution. Both support aggregation for queries that contain monotonic aggregate functions with a partially ordered domain as defined by Ross and Sagiv [25]. BOOM analytics [26] implemented the Hadoop MapReduce framework and HDFS storage system using Overlog whereas LogicBlox [27] is a commercial database system based on LogiQL.

VII. Conclusions

We introduce Datalography , the first complete Datalog system built on top of a think-like-a-vertex parallel graph processing platform, Apache Giraph, which is representative of a class of platforms based on the same paradigm (e.g. Pregel, GPS, GraphLog, GraphX).

Our approach exploits the alignment of the Datalog and think-like-a-vertex paradigms to show that no modifications of the platform are required. Therefore, the ideas and techniques presented here apply to all class members.

A remarkable feature of our solution is that it facilitates porting the actual code across Giraph-class platforms. This is due to the fact that the complexity resides in capturing the evaluation strategy and optimizations via Datalog program rewriting. The rewriting is think-like-a-vertex aware but is performed statically outside the vertex program. The rewriter code can therefore be reused without change. The only code that requires porting is the vertex program which is in charge of local rule evaluation and is relatively simple.

Depending on the application, certain developers would trade some performance for the conciseness and the abstraction from details afforded by Datalog (recall that the SSSP, WCC and PageRank analytics are expressed in up to 3 Datalog lines but require 60, 50, resp. 100 Java lines in the customized applications included in the Giraph distribution). The fact that our optimizations actually improve performance over custom imperative code comes as an added benefit.

References

[1] J. Seo, S. Guo, and M. S. Lam, "Socialite: Datalog extensions for efficient social network analysis," in *ICDE*, 2013.

[2] J. Seo, J. Park, J. Shin, and M. S. Lam, "Distributed socialite: A datalog-based language for large-scale graph analysis," *PVLDB*, vol. 6, no. 14, pp. 1906–1917, 2013.

[3] J. Wang, M. Balazinska, and D. Halperin, "Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines," *PVLDB*, vol. 8, no. 12, pp. 1542–1553, 2015.

[4] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo, "Big data analytics with datalog queries on spark," in *SIGMOD*, 2016.

[5] Y. Bu, V. Borkar, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan, "Scaling datalog for machine learning on big data," in *arXiv preprint*, no. 1203.0160, 2012.

[6] A. Ching, "Scaling apache giraph to a trillion edges," *Facebook Engineering blog*, 2013.

[7] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD*, 2010.

[8] S. Salihoglu and J. Widom, "GPS: a graph processing system," in *SSDBM*, 2013.

[9] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning in the cloud," *PVLDB*, vol. 5, no. 8, pp. 716–727, 2012.

[10] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *OSDI*, 2014.

[11] M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems," in *PVLDB*, 2015.

[12] F. Bancilhon and R. Ramakrishnan, "An amateur's introduction to recursive query processing strategies," in *SIGMOD*, 1986.

[13] D. B. Kemp and K. Ramamohanarao, "Efficient recursive aggregation and negation in deductive databases," *IEEE Trans. Knowl. Data Eng.*, vol. 10, no. 5, 1998.

[14] E. F. Codd, *Relational completeness of data base sublanguages*. IBM Corporation, 1972.

[15] W. P. Yan and P.-A. Larson, "Eager aggregation and lazy aggregation," in *VLDB*, 1995.

[16] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. R. Jr., "Query processing in a system for distributed databases (SDD-1)," *ACM Transactions on Database Systems*, vol. 6, no. 4, pp. 602–625, 1981.

[17] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[18] T. J. Ameloot, F. Neven, and J. V. den Bussche, "Relational transducers for declarative networking," in *PODS*, 2011.

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

[20] M. Maiterth, *Parallel Datalog on Pregel*. Institut fur Informatik der Ludwig-Maximilians-Universitat Munchen, October 2012.

[21] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012.

[22] A. Shkapsky, M. Yang, and C. Zaniolo, "Optimizing recursive queries with monotonic aggregates in deals," in *ICDE*, 2015.

[23] B. Chin, D. von Dincklage, V. Ercegovac, P. Hawkins, M. S. Miller, F. J. Och, C. Olston, and F. Pereira, "Yedalog: Exploring knowledge at scale," in *SNAPL*, 2015.

[24] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak, "Consistency analysis in bloom: a CALM and collected approach," in *CIDR*, 2011.

[25] K. A. Ross and Y. Sagiv, "Monotonic aggregation in deductive databases," in *PODS*, 1992.

[26] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears, "Boom analytics: exploring data-centric, declarative programming for the cloud," in *EuroSys*, 2010.

[27] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn, "Design and implementation of the logicblox system," in *SIGMOD*, 2015.