Ariadne: Online Provenance for Big Graph Analytics

Vicky Papavasileiou
University of California-San Diego
vpapavas@eng.ucsd.edu

Ken Yocum
Intuit, Inc.
University of California-San Diego
kyocum@eng.ucsd.edu

Alin Deutsch
University of California-San Diego
deutsch@eng.ucsd.edu

ABSTRACT

Data provenance is a powerful tool for debugging large-scale analytics on batch processing systems. This paper presents Ariadne, a system for capturing and querying provenance from *Vertex-Centric* graph processing systems. While the size of provenance from map-reduce-style workflows is often a fraction of the input data size, graph algorithms iterate over the input graph many times, producing provenance much larger than the input graph. And though current provenance tracing procedures support explicit debugging scenarios, like crash-culprit determination, developers are increasingly interested in the behavior of analytics when a crash or exception does not occur.

To address this challenge, Ariadne offers developers a concise declarative query language to capture and query graph analytics provenance. Exploiting the formal semantics of this datalog-based language, we identify useful query classes that can run *while* an analytic computes. Experiments with various analytics and real-world datasets show the overhead of `online` querying is 1.3x over the baseline vs. 8x for the traditional approach. These experiments also illustrate how Ariadne's query language supports execution monitoring and performance optimization for graph analytics.

ACM Reference Format:
Vicky Papavasileiou, Ken Yocum, and Alin Deutsch. 2019. Ariadne: Online Provenance for Big Graph Analytics. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3299869.3300091

1 INTRODUCTION

Big Graph (analogous to Big Data) processing engines like Giraph [8], GraphX [10], GraphLab [16] and Pregel [17] 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. Big 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 vertex-centric programming model where a single program repeatedly runs on each vertex.

Developers spend much of their time analyzing the behavior of their analytics. For instance, they look for aberrant algorithm behavior during code development or when using new data sets. This involves "crash culprit determination" – finding input data elements that caused code to fail. However, there are other equally important non-crash related activities. These include improving result quality and efficiency of computation, asserting behavior invariants, and checking for data formats and ranges.

Provenance can enable such exploration through tools that allow developers to analyze and query the way data changes during computation. Prior work on provenance for data-parallel workflows illustrates the power of pinpointing data inputs responsible for a system crash or exception [12, 13, 15, 21]. However, these approaches do not meet the needs of Big Graph analytics: none address provenance for Big Graph systems nor provide the ability to analyze behavior beyond crash-culprit determination.

Though *Vertex-Centric* graph programs are relatively simple to devise, they present challenges to effectively capturing and querying provenance. VC engines repeatedly execute the same code on each graph vertex for tens to hundreds of iterations. Our experiments show that the provenance of Big Graph analytics can be 10x larger than the input graph whereas the provenance of Spark [30] analytics amounts to 30% – 50% of the input [13]. 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.

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3300091

This paper presents Ariadne, a system to declaratively customize provenance capturing and querying for Big Graph analytics on VC engines. Ariadne defines a provenance query language (PQL) that developers use to i.) customize provenance capture to reduce time and space overheads, and ii.) query the provenance in more meaningful ways than tracing. This work demonstrates PQL provenance queries that can assert invariants in the behavior of the analytic, audit input data, and even suggest approximate optimizations that trade accuracy for speed.

Crucial to the usability of PQL are the theoretical underpinnings that enable two novel provenance query evaluation methods. We identify PQL query subclasses that allow both scalable *layered* evaluation and *online* evaluation. While layered evaluation scales traditional offline querying, online evaluation obviates the need for a separate capturing step, short-circuiting the traditional capture-first, query-offline approach. Online query evaluation occurs alongside the unmodified analytic; at the end of computation both the analytic result and the provenance query result exist.

To summarize, Ariadne's contributions include:

- A formal provenance model for VC graph computations and a simple, efficient physical representation of the resulting provenance graph.
- A declarative, concise Datalog-based language for customized capturing and querying of provenance.
- Two novel and scalable provenance query evaluation methods: layered and online evaluation.
- A system architecture that translates provenance query evaluation to ordinary vertex programs without modifying the graph processing engine itself.
- An evaluation on a Giraph-based [8] prototype compares the traditional approach of capture-first, query-offline to the layered and online methods. Ariadne's online queries exhibit an average overhead of $1.3x$ across various analytics and real-world graphs.

Ariadne's design choices are both user-friendly and admit implementations across other VC systems without conceptual hurdles. For instance, Ariadne performs provenance capture and querying transparently to the graph analytic, i.e. requiring no code changes by the analytic's developer. In addition, while many front-end languages (e.g., Java, C++, Scala, Datalog) are in use today to program VC graph analytics, Ariadne's provenance graph representation is independent of the native language specifying the graph analytic. It only assumes that this computation conforms to the VC paradigm.

2 BACKGROUND AND SYSTEM OVERVIEW

This section describes the *Vertex-Centric* (VC) graph programming model implemented by the majority of Big Graph processing systems. Ariadne manages provenance for Big Graph analytics implemented according to this model and executing on VC systems. Moreover, Ariadne itself follows the VC model and executes on the same system the graph analytics run so that both computation and provenance analysis happen in the same framework. Next, we describe an example that highlights how to analyze graph analytics with the goal of improving their performance.

2.1 Vertex-centric graph processing

Big Graph processing frameworks implement the low-level modules of parallel and distributed evaluation of graph analytics. They offer an API for developers to implement such analytics and an execution environment to run them. Popular graph processing systems, inspired by Pregel [17], pair the Bulk Synchronous Parallel (BSP) computation model with the VC programming model. Ariadne explores this model because of its wide adoption – one system, Giraph [8], is used in research and industry, and scales to trillion of edges [5].

BSP proceeds in a series of *supersteps* i.e., iterations each followed by a global barrier. In the VC programming model, the developer specifies a single vertex program describing the computation a vertex performs in one superstep. All vertices compute, in parallel, the same vertex program which consists of three stages. First, a vertex reads data sent from its neighboring vertices. Second, the computing vertex processes the received information and updates its state. Finally, it sends its new state to its neighbors. See Appendix A for an example vertex program for single-source shortest paths.

Most VC systems use a messaging API where vertices communicate via message exchange that happens between supersteps to ensure correctness. This means that messages sent during a superstep are visible to their destination vertices only at the next superstep. At the beginning of each superstep, the system only runs vertices that received messages. Thus the graph analytic terminates when no vertex receives a message. VC systems are typically main-memory engines that load the entire graph in memory (potentially distributed across the machines in a cluster).

2.2 Motivating scenario

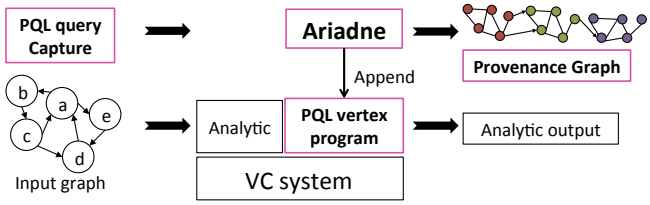
Prior work uses provenance in debugging scenarios to identify the causes of crashes. However, this covers only a small part of the tasks developers perform after developing a graph analytic. Specifically, they are interested in tuning tasks that improve the quality and/or efficiency of computation. Below we show an example of tuning enabled by Ariadne.

A common idea for iterative, fixpoint algorithms is to trade accuracy for speed by approximating the final result [19, 26]. 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. The idea is not new, implementations of PageRank in the libraries of VC systems are formulated this way. However, developers need to manually identify whether such an idea is applicable to other algorithms and manually identify the threshold denoting a large update or use expensive external tools [26].

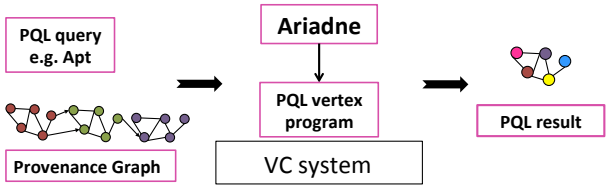
On the other hand, using Ariadne, a developer (Alice) can identify cases where this approximate optimization is possible and the potential gain to expect. Moreover, she can use the same query for different algorithms. We will call this the *a*pproximate o*pt*imization query or `apt`. We parameterize the `apt` query by a vertex value comparison function such as the difference or euclidean distance, and a convergence criterion, for instance a threshold $\epsilon = 0.001$. The query computes various metrics, such as how many vertices will not execute in a superstep with the given threshold. We show the query in Section 4.2 (Query 1).

The traditional approach for querying provenance first captures all provenance and then queries it offline. Ariadne supports and improves this scenario as Alice can now customize what information to capture, significantly reducing capture overheads. For example, the `apt` query refers only to the vertex values and not the message values, hence Ariadne does not need to capture those. Figure 1a shows Alice capturing provenance by issuing a declarative query alongside her *unchanged* graph analytic. Ariadne compiles this query into a provenance query vertex program and appends it to the analytic. Thus at every superstep a vertex evaluates both the graph analytic and the capturing query. At the end of computation, both the graph analytic's result as well as the custom provenance information (represented as a graph as explained in Section 3) exist. Then, Figure 1b shows how Alice uses Ariadne to analyze the captured provenance in the offline mode via the `apt` query. This time the VC system only evaluates Ariadne's query vertex program.

Besides improving traditional offline analysis, Ariadne enables a novel *online* provenance analysis method that short-circuits capturing and offline querying. Again, Ariadne appends the query vertex program to the analytic, as shown in Figure 2, only this time a vertex evaluates the graph analytic as well as the provenance query on the *transient* provenance graph. 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.



(a) Declarative provenance capturing.



(b) Declarative offline provenance querying.

Figure 1: Declarative capturing and offline querying using Ariadne.

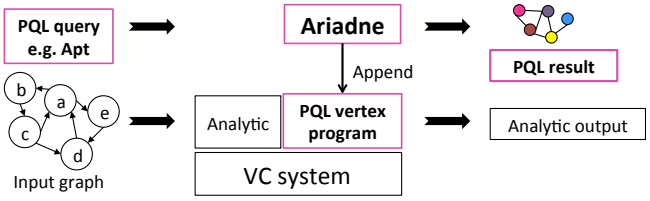


Figure 2: Online provenance querying using Ariadne.

Online provenance querying offers a shortcut to provenance management allowing Alice to validate various hypotheses quickly. Alice can evaluate multiple versions of the `apt` query to identify the threshold that gives the best performance versus accuracy tradeoff.

In the rest of the paper, we discuss the formal provenance model and query language semantics that enable scalable and efficient provenance querying. In the experiments section, we apply the `apt` query on PageRank, SSSP, WCC and ALS.

3 PROVENANCE GRAPH

Provenance of Big Graph analytics is multiple times larger than the input graph, comprising a Big Data problem in itself. Ariadne enables users to specify declaratively what information should be included in the provenance, thus customizing capturing. As we will see in Section 6, this offers great performance benefits.

We represent provenance as a graph that describes the analytic's computation from the perspective of vertices: It describes their values at every superstep, the messages they send and receive, and the values of their edges. A node in the provenance graph represents the execution of a vertex

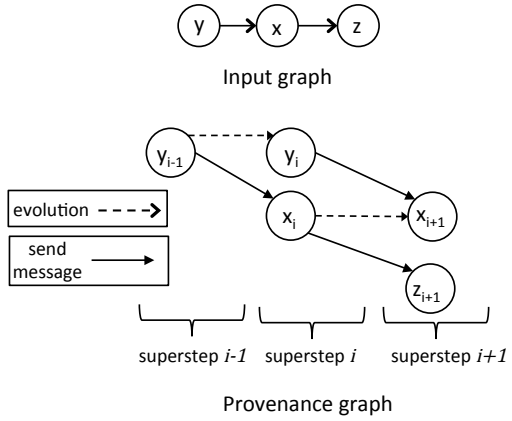


Figure 3: Input graph and corresponding provenance graph for SSSP analytic.

at a specific superstep. Note, in the remainder of this paper we will use *node* to refer to the vertices of the provenance graph to distinguish from the vertices of the input graph. Every provenance node is annotated with the value of the vertex at that superstep. There are two kinds of edges in the provenance graph: The first, the *send/receive message* edges, connect nodes that represent neighboring vertices in the input graph and shows the message exchange between them. The second, the *evolution* edges, connect nodes that represent the same vertex but at consecutive supersteps providing information about when a vertex was active and how its value evolves. An upper bound on the size of the provenance graph when all information is captured, is $n \times G_{in}$ where G_{in} is the input graph and n is the number of supersteps an analytic ran.

For example, consider the small input graph in the top of Figure 3 with vertex x, its incoming neighbor y and outgoing neighbor z. Assume Alice computes Single Source Shortest Paths (SSSP) and wishes to capture in the provenance when a vertex computed and when it sent messages to its neighbors because it updated its distance. 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 bottom of Figure 3 shows the provenance graph.

Observe that the provenance graph at superstep i contains a node for every input vertex that computed at that superstep. 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 input graph. Based on this observation, we propose a compact representation of the

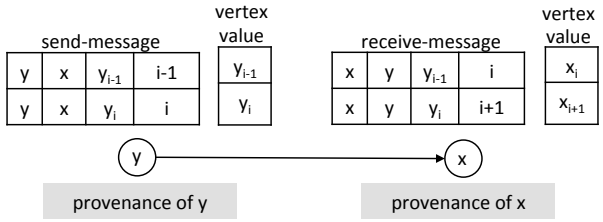


Figure 4: Compact provenance graph

provenance graph that consists of the input vertices and edges (or subset thereof) annotated with relations (tables) that contain the captured provenance information. Whereas in the unfolded provenance graph, we have multiple nodes representing the same vertex at different supersteps, in the compact format, we have one node with table annotations. These provenance tables are:

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

Figure 4 shows the compact representation of the provenance graph in Figure 3. 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 holds the value of each vertex for every superstep it was active in.

Both unfolded and compact provenance graphs contain the same information but the compact representation facilitates lower communication and memory overheads. If the provenance graph contains n nodes for n instantiations of a vertex, the compact format contains one node with n tuples. Accessing the different values of a vertex across all supersteps requires n supersteps whereas with the compact format it requires 1. Moreover, 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, exploiting the insight that nodes connected through evolution edges represent the same vertex at different instances in the computation and hence can be compacted.

4 PROVENANCE QUERY LANGUAGE

Ariadne provides a declarative front-end for users to capture and query the provenance graph. For this, Ariadne

incorporates a compiler that maps query evaluation to vertex programs. In this section, we present our Provenance Query Language (PQL), based on Datalog, and describe the necessary extensions to add distribution and ensure query evaluation conforms to the VC paradigm.

4.1 Standard Datalog

Recent work [20, 25, 28] has shown that Datalog can be used to express very succinctly graph and ML algorithms and compute them faster than their imperative counterparts. Datalog is a perfect fit for PQL as it is high-level, amenable to distributed and parallel evaluation and naturally expresses recursion.

A Datalog rule has the following syntax:

$$P_0 \leftarrow P_1, ..., P_n$$

where each P_i is a predicate of the form $P_i(v_1 \ldots, v_n)$. Since Datalog is a relational language, predicates denote relations (tables). Predicate P_0 is the head of the rule (also called IDB) and specifies the table holding the results of rule evaluation. The conjunction of predicates $P_1 \ldots P_n$ forms the body of the rule and specify input tables accessed during evaluation. Initially, the database contains a set of ground tables (EDBs) like the set of vertices and edges of a graph. These ground tables appear in the body of rules. Rule evaluation uses these tables to infer new user-defined tables (with the schema of IDB predicates) that are added to the database. A Datalog rule is read as: "if there exist tuples in the database for which $P_1 \ldots P_n$ are true, then P_0 is true as well". Recursion is expressed by appearance of the head predicate in the body of its defining rule.

A Datalog query is a collection of Datalog rules. The evaluation of each rule proceeds bottom-up, starting from ground tables and inferring new tuples for rules using Semi-Naive evaluation [4]. Moreover it is iterative, and continues until a fixed-point is reached where no new tuples can be inferred.

4.2 PQL syntax

PQL queries have access to the EDB predicates shown in Table 1, populated by provenance capturing. The relational tables that correspond to the EDB predicates are partitioned across the nodes of the provenance graph. We extend every PQL predicate with a first term that is the *location specifier*, introduced in [20], to denote the *location* of a partition, i.e. the node where a set of tuples resides. Thus, every predicate in Table 1 has as first term x.

Every predicate P_i for $i \in [1..n]$ in the body of a PQL rule can be either:

- one of the built-in primitive predicates from Table 1 (possibly negated),
- a positive relational predicate of the form $P_i(x, \overline{v})$ with P_i a relation name, x a location variable and $\overline{v}$ a vector of attribute variables, or
- a negated relational predicate of the form $\neg P_i(x, \overline{v})$ (using above notation),
- a comparison predicate of the form $t_1 \theta t_1$ where t_1, t_2 are terms (variables, constants, arithmetic expressions or function calls over them) and $\theta \in \{=, \neq, >, \geq, <, \leq\}$ is an arithmetic comparison operator, or
- a boolean function call $f(\overline{v})$ with f a function name (built-in or user-defined).

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

Table 1: Provenance EDB predicates

The head predicate, P_0, may be a positive predicate or of the form $P_0(x, \overline{v}, AGGR(t))$ with x a location specifier, variables $x, \overline{v}$ the grouping terms, *AGGR* the aggregation function (built-in or user-defined) and t the aggregating variable.

We follow the semantics of [29] and support in recursion stratified negation, and monotonic aggregates (min, max, sum, count) with respect to set-containment on the domain of positive numbers, as well as monotonic arithmetic (+,*,...) and boolean functions.

Using the above syntax, Alice would express the `apt` query using PQL as Query 1 where the first rule creates table **change** at a node x if the value d_1 of a vertex x at superstep i and its value d_2 at the preceding superstep j differ less than a threshold ϵ. The next rule, **neighbor-change**, is true if vertex x received at least one message from a neighbor whose value changed *more* than the threshold (large update). Rule **no-execute** determines if at superstep i vertex x would not execute given the current threshold. This is true if x did not receive any messages or if it received messages from neighbors with small updates in their values. Finally, rule **safe** checks whether it is safe for x to not execute at superstep i which is true if the difference in its value, had it executed, is smaller than the threshold. Rule **unsafe** contains the vertices for which it is not safe to skip their execution because they experience large updates.

We see that the vocabulary of PQL is parsimonious, comprising high-level primitives that describe a vertex-centric

change$(x, i) \leftarrow$ value(x, d_1, i), value(x, d_2, j),
evolution(x, j, i), udf-diff(d_1, d_2, ϵ).
neighbor-change$(x, i) \leftarrow$ receive-msg(x, y, m, i),
$\neg$ change$(y, j), j = i - 1$.
no-execute$(x, i) \leftarrow \neg$ neighbor-change(x, i), superstep(x, i).
safe$(x, i) \leftarrow$ no-execute(x, i), change(x, i).
unsafe$(x, i) \leftarrow$ no-execute$(x, i), \neg$ change(x, i).

Query 1: Approximate optimization query.

execution (e.g., the vertex value, the messages sent at a superstep, etc), while abstracting away the low-level details of the original graph analytic code in order to make PQL queries *independent* of the native language of the graph analytic. Therefore one could argue that PQL is more user-friendly than the native language of the graph analytic (the typical option in prior work). To enhance user-friendliness, follow-up work can address templates for PQL rules, or a GUI query builder with automatic translation to Datalog, inspired by the (tree-)pattern queries developed for XML and semistructured graphs.

4.3 Distributed semantics

Here, we focus on how we extend standard Datalog semantics to account for distribution and to conform to the VC paradigm. We observe that there is a correspondence between how Datalog queries are evaluated until no new tuples can be inferred and how VC graph analytics evaluate until no more messages exist in the system.

Initially, the distributed database contains only the EDBs of Table 1, partitioned over the graph nodes such that tuples are located at node x if and only if their location specifier is set to x. Rule evaluation uses these EDBs to infer new user-defined tables (the IDBs) that are added to the database, also partitioned across nodes.

Rule evaluation proceeds in *evaluation steps*. Every step consists in evaluating all rules simultaneously over the current distributed database snapshot. A *satisfying valuation* ν of a rule is a satisfying valuation of every predicate P_i for $i \in [1..n]$ in its body. ν maps P_i's variables to constants in the database that make P_i true. The tuples that result from a valuation of P_i are stored in a relational table named R_i. In our distributed setting, R_i is partitioned across graph nodes and its contents at a node $\nu(x)$ consists of the set of tuples located at $\nu(x)$ (whose location specifier is the valuation of x), denoted as $[\![R_i]\!](\nu(x))$. Every evaluation step computes first all satisfying valuations for the rule body at each node $\nu(x)$ (simultaneously) and then, modifies the contents of $[\![R_i]\!](\nu(x))$ in parallel. The query result is obtained by iterating evaluation steps until a fix-point is reached.

Rules may refer to *remote* predicates in their body, denoted by a different location specifier than that of the head. For such rules to be evaluated, communication must occur between the nodes in the body of a rule and its head. Consider rule r with head $P_0(x, \overline{v_0})$ and a relational predicate $P_i(y, \overline{v_i})$ in its body, and let ν be a satisfying valuation for r. If $\nu(x) \neq \nu(y)$ then, table R_i is remote and every node $\nu(y)$ needs to send its partition $[\![R_i]\!](\nu(y))$ in a message to node $\nu(x)$.

The VC paradigm allows communication between nodes that are directed neighbors. We say that a rule is *VC-compatible* if, whenever the location variable y of a body predicate is different from the location variable y of the head, y, x co-occur in a **send**/**receive-message** predicate that applies only to neighbors.

More formally, for a rule to be VC-compatible it must follow the VC normal form 4.1:

Definition 4.1.

$$r(x, \bar{v}) \vdash P_x(x, \bar{\chi}), \quad (1)$$
$$receive - message(y, x, i, m), \quad (2)$$
$$P_y(y, \bar{\psi}), \quad (3)$$
$$send - message(x, z, i, m), \quad (4)$$
$$P_z(z, \bar{\zeta}), \quad (5)$$

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

According to which a rule evaluated at a provenance node x has access to its local predicates P_x (line 1), the neighbors it received messages from (line 2) and their predicates P_y (line 3), and the neighbors it sent messages to (line 4) and their predicates P_z (line 5).

We say that a query is VC-compatible if all its rules are VC-compatible. For VC-compatible queries, the evaluation step corresponds to a superstep in the VC paradigm, with communication occuring between supersteps.

5 PQL EVALUATION

Ariadne uses PQL to declaratively customize capturing and querying of the provenance graph. Our experiments show that complete provenance graph capture is resource intensive, incurring 3-5x runtime overheads while the size of the provenance graph scales with superstep count (Section 6.1). Thus, loading the provenance graph for offline querying may not be possible on the same compute cluster used to run the analytic (as VC systems often materialize the graph in memory). This section presents two novel and efficient PQL evaluation methods that allow Ariadne to generate and query provenance from analytics without increases in compute resources.

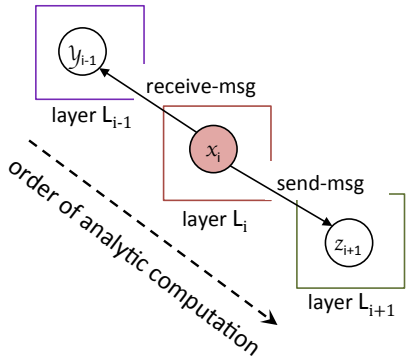


Figure 5: Provenance graph with layer annotations.

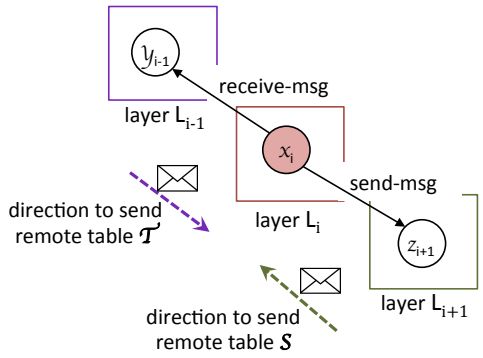


Figure 6: Rule R_1 requires both prior $(i-1)$ and successive layers $(i+1)$ to be present during evaluation.

5.1 Layered evaluation

We identify a class of PQL queries that don't require materializing the entire provenance graph but rather can evaluate on layers of it. First, let us define the layers of a provenance graph:

Definition 5.1 (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 the highlighted provenance node x_i in the provenance graph in Figure 5. The graph comprises 3 layers, one for each superstep the analytic computed. The provenance graph shows that at superstep i vertex x received a message from y (y sent the message at superstep $i-1$ according to the *Vertex-Centric* model). During the same superstep, x sent a message to vertex z, which received the message at superstep $i+1$. Note, that the send/receive-message edges cross layer boundaries.

Layered PQL query evaluation allows Ariadne to successively materialize individual layers of G_{pr}. This is because only a single layer's nodes execute at each superstep. For this to be possible, the order in which query evaluation visits the layers of the provenance graph must be the same as the direction in which nodes exchange messages with remote tables.

Below we show how one can syntactically infer when layered evaluation is possible and what the evaluation order should be. In general a PQL query must allow an evaluation ordering on the layers of the provenance graph such that: i) when the nodes in a layer evaluate a query, they have received all remote messages from their neighbors and ii) those neighbors evaluated in a previous layer according to this ordering.

We first illustrate with a counter-example rule that cannot be evaluated in a layered fashion due to the presence of both **send** and **receive-message** predicates. Assume 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 has location specifier x meaning it is evaluated by a provenance node x. The **receive-message** predicate defines the neighbors y from which x received messages at superstep i. Similarly, the **send-message** predicate refers to neighbors z to which x sent messages at superstep i. In addition, the rule uses two remote tables, **T** and **S**, located at neighbors y and z respectively. During query evaluation, neighbors y and z must send their tables to node x. Figure 6 illustrates node x_i receiving these remote tables from a prior (L_{i-1}) and successive (L_{i+1}) layer during evaluation. Hence we cannot impose an ordering on the layers of the provenance graph that satisfies both conditions i and ii above.

On the other hand, consider rule

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

that refers *only* to neighbors y from which x, the evaluating node, received messages at superstep i. During query evaluation, node y_{i-1} in layer L_{i-1} sends table **T** to x_i. It is easy to see that for queries like R_2 whose rules use only **receive-message** edges, messages are sent to successive layers of the provenance graph. Moreover, once the nodes in a layer have computed, they will never receive messages again thus will never compute again. Hence, query evaluation can proceed in an ascending order from layer 0 to layer n.

Similarly rule

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

refers *only* to neighbors z that x sent a message to at superstep i. During query evaluation, node z_{i+1} in layer L_{i+1} sends

table S to x_i. Like before, messages are sent unidirectional, this time to preceeding layers. We can now employ layered evaluation by visiting the layers of the provenance graph in descending order, starting from layer n to layer 0.

Thus, we define the *directed* PQL queries as follows:

Definition 5.2 (Directed). A PQL query Q is directed if for all rules in Q the variables of remote predicates in their bodies appear in either **send-message** predicates or **receive-message** predicates but not both.

We call queries whose rules use only **receive-message** edges *forward* PQL queries whereas we call queries whose rules use only **send-message** edges *backward* queries. Layered evaluation is guarded by n, where n is the number of supersteps the graph analytic ran.

Lemma 5.3. *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. □

Directed PQL queries are amenable to offline layered evaluation where the provenance graph layers are materialized sequentially, reusing the working memory, instead of materializing the entire graph at once. This enables scalable and efficient evaluation which significantly outperforms the entire-graph materialization approach, as demonstrated in our experiments.

5.2 Online evaluation

Forward PQL queries provide an additional opportunity to improve evaluation efficiency. In particular, one can evaluate this query class simultaneously with the graph analytic. This *online* provenance querying provides a shortcut to the traditional capture-first, query-offline approach, reduces runtime overheads from $2-8x$ to $1.3x$ (Section 6.2), and leaves the original vertex program code unchanged.

To see why this is possible, recall that forward query evaluation visits the provenance graph layers in ascending order – that order is identical to the chronological order of analytic computation. Capitalizing on this, Ariadne evaluates forward queries *online* alongside the graph analytic so that at the end of computation the results of both the analytic and PQL queries exist. To this end, Ariadne appends the PQL query to the original graph analytic such that at every superstep, a vertex evaluates both its vertex program and the PQL query. Moreover, Ariadne appends the query tables to the messages the vertices exchange.

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 5.4. *Let A be a graph analytic, Q a forward PQL query, G the input graph and G_{PR} the provenance graph 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 of query evaluation.
ii) The definition of forward queries specifies that remote predicates in the body of a rule must be guarded by **receive-message** predicates. Hence, message exchange during query evaluation can only happen between vertices that exchanged messages during analytic computation. □

6 EXPERIMENTAL EVALUATION

Our experiments show that Ariadne 1) reduces space and time overheads of provenance capturing 2) improves provenance querying performance with layered and online evaluation and 3) supports forward and backward lineage queries as well as novel provenance analysis queries not seen in prior work.

We experiment with 4 graph analytics, PageRank, SSSP, WCC and ALS. For each analytic, we evaluate 2 capturing queries (Queries 2, 3), 3 forward queries (different for each analytic) and 2 backward queries (Queries 10, 12). The forward queries use 3 different evaluation modes: online where no capturing is performed, offline using layered evaluation on the previously captured provenance graph, and offline that materializes the entire provenance graph. We ran each query 5 times and reported the trimmed mean, removing the shortest and longest runs.

We summarize the outcomes across all datasets and analytics (details are below). Capturing the full provenance takes $2.7x-5.6x$ the baseline (the original analytic's runtime

T), whereas customized capturing takes less than $2x$ T. Online querying takes $1.3x$ T, versus $3.5x$ T for offline layered evaluation. Moreover, backward tracing using offline layered evaluation on the full provenance graph takes $2.4x - 3.5x$ T, while offline layered evaluation on a custom provenance graph takes $0.5x$ T.

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 and results were all stored in HDFS with a replication factor of 2.

Algorithms and Datasets: We evaluate provenance capturing and querying on three well-known graph analytics popular in research and practice [23].

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 2. Note, we assigned random positive weights in the range of $0 - 1$ to the edges of the input graph for SSSP.

Table 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 users 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.

6.1 Provenance capturing

Prior work captures the entire provenance graph and stores it for offline querying. We simulate this scenario with Query 2 by way of capturing the values of a vertex at every superstep and the messages they send and receive. Table 3 shows the space overhead of the full provenance graph. For PageRank and SSSP the captured provenance is consistently $10x$ larger than the input graph whereas for WCC it is $5x$.

value$(x, v, i) \leftarrow$ vertex-value(x, v), superstep(x, i).
send-message$(x, y, m, i) \leftarrow$ send(x, y, m), superstep(x, i).
receive-message$(x, y, m, i) \leftarrow$ receive(x, y, m),
superstep(x, i).

Query 2: Capture full provenance graph

Table 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, Ariadne allows a user to customize provenance capturing to her needs, reducing space and time overheads. Consider Query 3 that captures information about the set of vertices influenced by an input vertex. This information suffices to answer forward tracing provenance queries. We ran Query 3 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 4 shows the size of the custom provenance graph which is always less than 40% of the input graph and contains more than 80% of the input vertices.

fwd-lineage$(x, v, i) \leftarrow$ value(x, v, i), superstep(x, i),
$x == \alpha, i == 0$.
fwd-lineage$(x, v, i) \leftarrow$ receive-message(x, y, m, i),
fwd-lineage(y, j, w), value(x, v, i).

Query 3: Capture custom provenance graph

Table 4: Size comparison of input graph and custom 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

[1] http://www.dis.uniroma1.it/challenge9/download.shtml
[2] http://grouplens.org/datasets/movielens/20m/

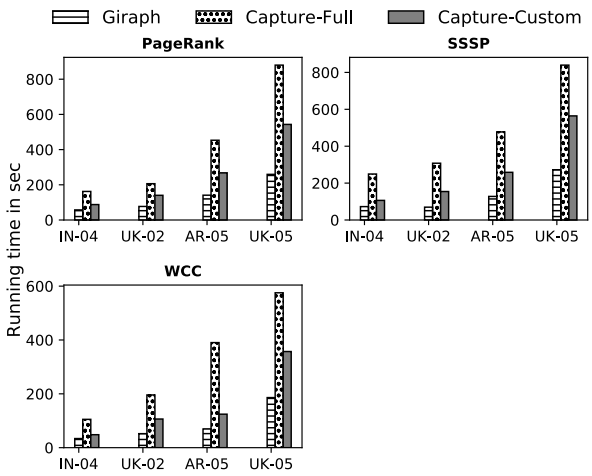


Figure 7: Runtime comparison of provenance capturing queries 2 (Full) and 3 (Custom).

Figure 7 shows the runtime overhead for capturing the full versus custom provenance graph. We see that Query 2 takes $2.7x - 3.4x$ PageRank and $3x - 5.6x$ SSSP or WCC. Query 3 takes less than $2x$. When the provenance graph exceeds the size of available RAM, Ariadne offloads it asynchronously 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. While this was due to a lack of memory allocation backpressure in our prototype, further research could improve support for such sizes and provenance generation rates.

6.2 Forward provenance querying

We evaluate the performance of three PQL query evaluation modes and compare them against the baseline of a graph analytic running on Giraph without any provenance capturing or querying (Giraph). We report the performance of online provenance querying (*Online*) that happens simultaneously with the analytic, layered offline querying (*Layered*) and straightforward offline querying (*Naive*) on the captured provenance graph. The running times reported for offline querying do not include the capturing overheads. Note how *Naive* was not able to scale beyond the two smallest datasets in any of our experiments.

6.2.1 Execution monitoring. Developing Big Graph analytics is an iterative process where developers constantly refine their code and data to improve the quality of outcomes. Ariadne enables developers to monitor the execution of their analytics (e.g. as new data arrives or parameters change) and quality of their data (e.g. sanity check or anonymization). Developers benefit from online evaluation where these PQL queries can be always "on", in the sense that they are part of every run of an analytic. For that, it is imperative for them to incur minimal overhead.

PageRank Query 4 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. In Figure 8 top row Query 4 takes $1.14x$ PageRank using *Online* while it takes $3x$ PageRank with *Layered* and $4x$ PageRank with *Naive*.

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

Query 4: PageRank

SSSP and WCC Query 5 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 8 middle row shows the runtime for SSSP on the left: Query 5 takes $1.13x$ SSSP using *Online*, $3.5x$ SSSP using *Layered* and $4.6x$ SSSP using *Naive*. The running time overheads for WCC on the right are similar.

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

Query 5: SSSP and WCC

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 6 ensures that if a vertex received no messages, then its value doesn't change. Figure 8 last row on the left shows the query incurs $1.3x$ SSSP using *Online*, $3.6x$ SSSP using *Layered* and $4.7x$ SSSP using *Naive*. For WCC, on the right, the query takes $1.2x$ WCC using *Online*, $3.7x$ WCC using *Layered* and $4.3x$ WCC using *Naive*.

Notice, how we used the same queries for both analytics. PQL provides a common front-end to developers to query different analytics, saving them from duplicate manual efforts.

ALS Query 7 checks that the local error for every vertex is between the range of $0 - 5$ which is the range of the ratings

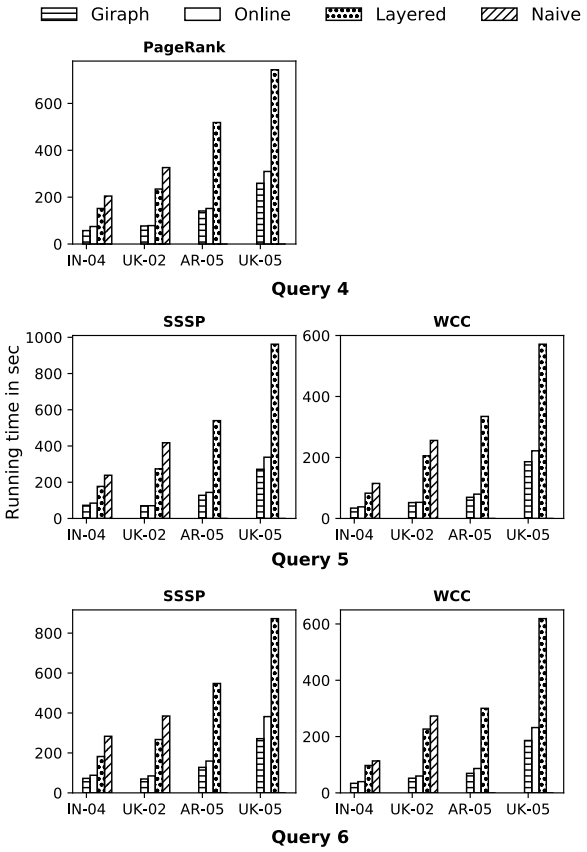


Figure 8: Runtime of execution monitoring queries 4, 5 and 6.

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

Query 6: SSSP and WCC

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 9 left shows the query adds 5% overhead using *Online*.

input-failed$(x, y, i) \leftarrow$ prov-error(x, y, i, e),
edge-value$(x, y, w, i), e < 0, e > 5, w < 0, w > 5$.
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 7: ALS

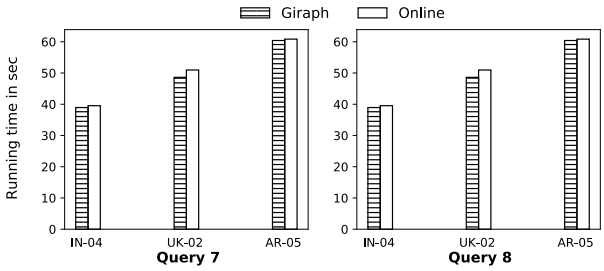


Figure 9: Running time for ALS queries.

Query 8 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 9 right shows the query takes $1.2x$ ALS using *Online*.

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

Query 8: ALS

6.2.2 Performance tuning. We now turn to our motivating scenario where developers use provenance to tune the performance of their graph analytics. We measure the overhead of our motivating Query 1, copied here for ease of readability. Moreover, we show that insights gained from using the query on one dataset, are transferable to unseen datasets.

We use the same query for all our analytics by parameterizing it with different thresholds (ϵ) and vertex value comparison functions (udf-diff). 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 returns the vertices whose execution can be safely skipped given the threshold. The larger the fraction of safe versus unsafe vertices, the smaller the error of approximation.

We measure the error of approximation in the same manner as [26] 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.

change$(x, i) \leftarrow$ value(x, d_1, i), value(x, d_2, j),
evolution(x, j, i), udf-diff(d_1, d_2, ϵ).
neighbor-change$(x, i) \leftarrow$ receive-msg(x, y, m, i),
$\neg$ change$(y, j), j = i - 1$.
no-execute$(x, i) \leftarrow \neg$ neighbor-change(x, i), superstep(x, i).
safe$(x, i) \leftarrow$ no-execute(x, i), change(x, i).
unsafe$(x, i) \leftarrow$ no-execute(x, i), $\neg$ change(x, i).

Apt query (motivating example)

Figure 11 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 60% of the vertices can safely skip their execution for 10 out of 20 supersteps and that there are no unsafe vertices. The optimization is already part of some PageRank implementations, here we use the results as a proof of concept. The error when computing the optimized PageRank is shown in Table 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 10 left is $1.4x$. The runtime in Figure 11 top left takes $1.3x$ PageRank using *Online*, $3.2x$ PageRank using *Layered* and $3.8x$ PageRank using *Naive*.

For SSSP and $\epsilon = 0.1$ the query finds that 87% of the input vertices can safely skip their execution for more than 2 supersteps and 11% of them can do so for more than 10 consecutive supersteps. Moreover, there are no unsafe vertices. The query takes $1.5x$ SSSP using *Online*, $3.5x$ SSSP using *Layered* and $5x$ SSSP using *Naive* (Figure 11 top right). Again the results lead a developer to incorporate the optimization to SSSP. Table 6 shows the error across all datasets is 10^{-2} when using the same threshold. The speedup in Figure 10 right is $1.8x$ the baseline.

With WCC we used a threshold of 1 since the amount of difference between component IDs doesn't matter, but rather the fact that vertices are assigned to different components. Although the query finds vertices that would not execute because their neighbors changed less than threshold, they can never do so safely. In other words, all vertices that belong in table **no-execute** are also part of table **unsafe** whereas table **safe** is empty. 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

Table 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

Table 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

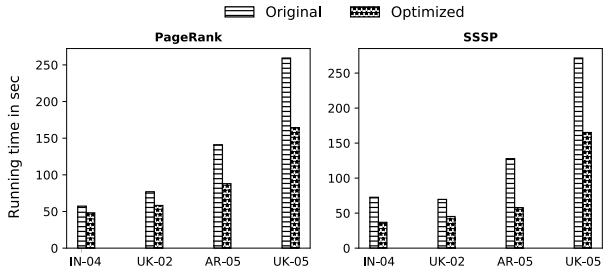


Figure 10: Runtime improvement between original and optimized analytic

across all datasets. Figure 11 bottom left shows the query takes $1.6x$ WCC using *Online*, $3.6x$ WCC using *Layered* and $5x$ WCC using *Naive*.

Finally, for ALS the query returns too few vertices for both **safe** and **unsafe** tables suggesting the requirement of a more fine-tuned convergence criterion. As future work, we plan to do a user study with developers using Ariadne to write algorithm-aware tuning queries. Nevertheless, the overhead for ALS in Figure 11 bottom right is always lower than 10%.

6.3 Backward provenance querying

A common provenance operation is backward tracing to identify the input data items that lead to an item in the output. Since provenance querying follows the opposite order of computation, backward queries are evaluated offline and necessitate capturing of the provenance graph. Below we compare two approaches and report the runtime as an overhead of the original analytic: i) Capture the entire provenance

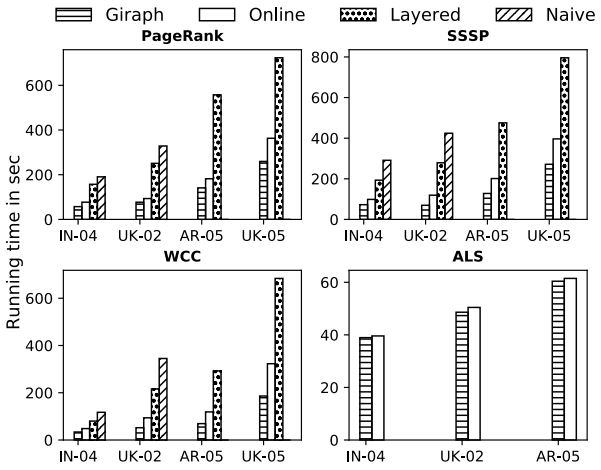


Figure 11: Runtime of motivating example Query 1

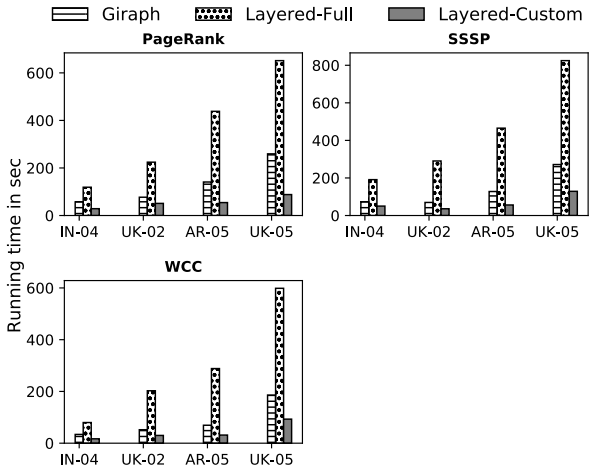


Figure 12: Runtime of layered backward querying using Query 10 (Full) and Query 12 (Custom)

graph using Query 2 and perform backward querying using Query 10 that entails $2.6x - 3.4x$ overhead and ii) Capture a custom provenance graph using Query 11 and query it using Query 12 that incurs $0.5x$ overhead. In both cases, offline querying is performed in a layered fashion. Notice, the result of the backward queries is the *lineage* of a vertex in the output.

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

Query 10: Backward lineage on full provenance graph

The first part of rule **back-trace** in Query 10 specifies the starting vertex α and superstep σ. The second part traces the provenance graph using the **send-message** edges. The rule is recursive and transitively visits all vertices in the subgraph rooted at the starting vertex. Finally, rule **back-lineage** contains the vertices active at superstep 0 that are reached by the transitive closure.

Notice that the query does not access the values of the messages that were sent but only the information of whether a message was sent. Moreover, for analytics where vertices send messages to all their outgoing neighbors (instead of a dynamic subset) it is not necessary to use the **send-message** edges of the provenance graph since the same information is encoded in the **edges** of the input graph. Ariadne allows developers to take advantage of this information to customize the captured provenance graph. We already discussed in Section 6.1 the performance benefits Ariadne provides to capturing. Here, we focus on the performance benefits it entails to offline querying.

Query 11 captures a custom provenance graph that does not contain the message values nor the send-message edges. 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. Then, updating the backward lineage query is easy. One simply needs to replace relation **send-message** with relations **prov-send** and **prov-edges** to obtain Query 12. The result of the query contains the exact same information as Query 10 but is $2x - 3x$ faster.

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

Query 11: Custom provenance for backward query

back-trace$(x, i) \leftarrow$ prov-value$(x, i, v), i = \sigma, x = \alpha$.
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 12: Backward lineage on custom provenance

Below, we compare the runtime of layered offline evaluation using the full provenance graph (*Full*) against using the custom 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.

Full is 2.6*x* PageRank while *Custom* takes only 0.5*x* PageRank. For SSSP, *Full* takes 3.4*x* over the baseline while *Custom* takes 0.5*x*. For WCC the overheads are similar. This highlights the strength of customized 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.

7 RELATED WORK

Although provenance querying has been studied in the context of databases [9, 14], scientific workflows [2, 7] and large-scale distributed engines [6, 13], 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 [9], 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 developers 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 [24], 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 [15] and Ramp [21] that capture provenance for MapReduce workflows on Hadoop [3] and offer offline tracing using external tools. Titian [13] 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 1.4*x* 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 10*x* 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 [22, 27, 31] to improve Ariadne's performance when capturing the full provenance graph.

[1] tracks How-Provenance [11] 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).

[6] addresses backward provenance tracing on a differential dataflow system [18] 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, developers can apply provenance pruning both to capturing (capture only the top-k data items) and to querying by customizing their queries.

8 CONCLUSION

This paper presents Ariadne, a declarative language and query evaluation system for capturing and querying provenance on Big 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 that developers can use to mine provenance, enabling analysis beyond crash-culprit determination. PQL supports invariant checks on data and computation to ensure correct execution, detecting outliers in algorithm behavior or in the data, and analyzing the runtime behavior of approximate analytics. And by offering online evaluation, Ariadne enables developers to cheaply add always-on checks to graph analytics. This work is a step towards viewing provenance as a run-time asset, not just a retrospective tool, for accurate and efficient Big Graph analytics.

ACKNOWLEDGMENTS

The authors thank our anonymous reviewers and shepherd, Daniel Deutch, for their insightful comments. In addition, we thank Wojciech Kazana who contributed to the early development of PQL. This work was supported by the Natural Science Foundation under Grant No. 1219220.

REFERENCES

[1] 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.

[2] Manish Kumar Anand, Shawn Bowers, and Bertram Ludäscher. Techniques for efficiently querying scientific workflow provenance graphs.

In *EDBT*, volume 10, pages 287–298, 2010.
[3] Apache Hadoop. Apache Hadoop. http://hadoop.apache.org, 2018.
[4] François Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *SIGMOD*, pages 16–52, 1986.
[5] Avery Ching. Scaling apache giraph to a trillion edges. *Facebook Engineering blog*, 2013.
[6] Zaheer Chothia, John Liagouris, Frank McSherry, and Timothy Roscoe. Explaining outputs in modern data analytics. *PVLDB*, 9(12):1137–1148, 2016.
[7] Yingwei Cui and Jennifer Widom. Lineage tracing for general data warehouse transformations. *PVLDB*, 12(1):41–58, 2003.
[8] Apache Giraph. Apache Giraph. https://giraph.apache.org/, 2018.
[9] Boris Glavic and Gustavo Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE*, pages 174–185, 2009.
[10] 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*, 2014.
[11] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.
[12] Muhammad Ali Gulzar, Matteo Interlandi, Tyson Condie, and Miryung Kim. Debugging big data analytics in spark with *BigDebug*. In *SIGMOD*, pages 1627–1630, 2017.
[13] 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.
[14] Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Querying data provenance. In *SIGMOD*, pages 951–962, 2010.
[15] Dionysios Logothetis, Soumyarupa De, and Kenneth Yocum. Scalable lineage capture for debugging DISC analytics. In *SOCC*, pages 17:1–17:15, 2013.
[16] 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.
[17] 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*, pages 135–146, 2010.
[18] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *CIDR*, 2013.
[19] Ioannis Mitliagkas, Michael Borokhovich, Alexandros G. Dimakis, and Constantine Caramanis. Frogwild!: Fast pagerank approximations on graph engines. *PVLDB*, 8(8):874–885, 2015.
[20] Walaa Eldin Moustafa, Vicky Papavasileiou, Ken Yocum, and Alin Deutsch. Datalography: Scaling datalog graph analytics on graph processing systems. In *Big Data*, pages 56–65. IEEE, 2016.
[21] Hyunjung Park, Robert Ikeda, and Jennifer Widom. RAMP: A system for capturing and tracing provenance in mapreduce workflows. *PVLDB*, 4(12):1351–1354, 2011.
[22] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488, 2013.
[23] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *PVLDB*, 11(4):420–431, 2017.
[24] 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.
[25] Jiwon Seo, Stephen Guo, and Monica S. Lam. Socialite: Datalog extensions for efficient social network analysis. In *ICDE*, pages 278–289, 2013.
[26] Zechao Shang and Jeffrey Xu Yu. Auto-approximation of graph computing. *PVLDB*, 7(14):1833–1844, 2014.
[27] 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.
[28] 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.
[29] Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. Optimizing recursive queries with monotonic aggregates in deals. In *ICDE*, pages 867–878, 2015.
[30] 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*, 2012.
[31] 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.

A VERTEX-CENTRIC PROGRAMMING MODEL

Ariadne is independent of the language of the graph analytic as long as it conforms to the *Vertex-Centric* (VC) programming model. In fact, many front-end languages can be used to program VC graph analytics, ranging from imperative (Java, C++, Scala) to declarative (Datalog). The Ariadne design admits the implementation to be ported to other VC systems without conceptual hurdles as neither the graph analytic nor the VC system are changed.

Below, we describe the general structure of a VC analytic and provide the pseudo-code corresponding to an imperatively-specified vertex program that implements Single-Source Shortest Path according to the VC programming model. The Giraph distribution [8] includes this (and many other algorithms) programmed in Java.

Algorithm 1 Pseudocode of VC graph analytics

```
1: function VertexProgram(v)
2:     Read messages from incoming neighbors of v.
3:     Update vertex v's value.
4:     Send messages to outgoing neighbors of v.
5: end function
```

In the VC model, computation proceeds in supersteps and all vertices compute in parallel the same vertex program. A vertex computes only if it has received messages, and computation stops when no more messages exist in the system. A general vertex program, see Algorithm 1, comprises three basic steps: First, a vertex reads all the messages it received from it incoming neighbors. Then, it updates its value performing computation based on its current state and the message it received. Third, it sends messages to its outgoing neighbors.

Algorithm 2 illustrates the implementation of SSSP. 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. Lines 10-11 select 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 14 updates the value of the current vertex. Then, lines 15-17 send messages to the outgoing neighbors with the new distance plus the respective edge weight.

Algorithm 2 SSSP implementation

```
1:  function SSSP(vertex,messages)
2:      if  superstep == 0 then
3:          vertex.value = MAX.DOUBLE
4:      end if
5:      if isSource(vertex) then
6:          minDist = 0
7:      else
8:          minDist = MAX.DOUBLE
9:      end if
10:     for m : messages do
11:         minDist = Math.min(minDist, m)
12:     end for
13:     if minDist < vertex.value() then
14:         vertex.value = minDist
15:         for  e: vertex.out-edges do
16:             distance = minDist + e.weight
17:             sendMessage(e.target, distance)
18:         end for
19:     end if
20: end function
```