Efficient Runtime Capture of Multiworkflow Data Using Provenance

Renan Souza[§,°], Leonardo Azevedo[§], Raphael Thiago[§], Elton Soares[§], Marcelo Nery[§]
Marco A. S. Netto[§], Emilio Vital Brazil[§], Renato Cerqueira[§], Patrick Valduriez[#], Marta Mattoso[°]

§IBM Research, Rio de Janeiro, Brazil
°COPPE/Federal University of Rio de Janeiro, Brazil
#Inria & LIRMM, U. Montpellier, France

***Abstract*—Computational Science and Engineering (CSE) projects are typically developed by multidisciplinary teams. Despite being part of the same project, each team manages its own workflows, using specific execution environments and data processing tools. Analyzing the data processed by all workflows globally is a core task in a CSE project. However, this analysis is hard because the data generated by these workflows are not integrated. In addition, since these workflows may take a long time to execute, data analysis needs to be done at runtime to reduce cost and time of the CSE project. A typical solution in scientific data analysis is to capture and relate the data in a provenance database while the workflows run, thus allowing for data analysis at runtime. However, the main problem is that such data capture competes with the running workflows, adding significant overhead to their execution. To mitigate this problem, we introduce in this paper a system called ProvLake, which adopts design principles for providing efficient distributed data capture from the workflows. While capturing the data, ProvLake logically integrates and ingests them into a provenance database ready for analyses at runtime. We validated ProvLake in a real use case in the O&G industry encompassing four workflows that process 5 TB datasets for a deep learning classifier. Compared with Komadu, the closest solution that meets our goals, our approach enables runtime multiworkflow data analysis with much smaller overhead, such as 0.1%.**

I. Introduction

Computational Science and Engineering (CSE) projects are typically developed by multidisciplinary teams, each managing its own workflows with specific execution environments and data transformation tools. Each workflow processes (consuming and generating) large amounts of complex and heterogeneous data. Analyzing the data transformed by all workflows globally allows for understanding each data transformation by monitoring, debugging, and inspecting input and output datasets while workflows run, *i.e.*, at runtime—which is necessary to reduce cost and time of the CSE project. However, this analysis is hard because the data generated by these workflows are not related to their data transformations, which also impacts such relationship determinations.

To illustrate, consider an example of a CSE project, our case study (Figure 1), whose goal is to deliver deep learning models with high quality for an application in the Oil and Gas (O&G) industry. These four workflows generate data that are implicitly related through their data transformations but analyzing them globally after the data have been generated is complex because each data store is distributed with no information on the data transformations or how to relate the stores. There is no point in moving and integrating all data in a single repository for a global analysis. However, a complementing data representation on how the data in the data stores relate to each other contributes to a logically integrated multiworkflow data analysis, while keeping the autonomy of each data store. In this example (Figure 1), there is also the challenge of relating data from heterogeneous representations.

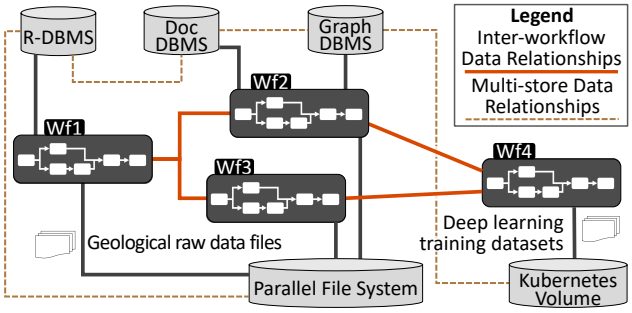

Fig. 1. Four workflows using five data stores.

A typical solution in scientific data analysis is to capture and relate the data in a provenance database at runtime [1]–[4]. Provenance data representation has a W3C recommendation, PROV [5], which has been used as a reference model to represent relationships between datasets and their data transformations in workflows. PROV-based databases follow a uniform way of representing "consumed" and "generated" data relationships between datasets and their data transformations, and other workflow data relationships. However, the main problem in runtime data capture is that it competes with the running workflows, adding significant overhead to their execution.

One exception in low overhead provenance data capture is DfAnalyzer [4], however this approach limits its analysis to isolated workflows, characterizing a *single-workflow* data capture solution. Also, its data capture is limited to raw data in file systems, unlike the data stores in Figure 1. In typical CSE projects, the execution autonomy of each single-workflow participating in a multiworkflow prevents data capture to be managed by a single-workflow data capture solution.

Some limitations in single-workflow data capture solutions are caused by the lack of: (i) capturing and relating data from autonomous workflow executions; (ii) globally identifying data to establish relationships from multiworkflow data (*i.e.*, data processed by a multiworkflow) in multiple stores; and (iii) overhead management strategies for capturing provenance data from one workflow while capturing data from another workflow potentially running in parallel.

The closest solution to meet our goals, Komadu is a distributed data capture solution that integrates provenance data in a multiworkflow execution [6,7]. Komadu captures provenance data generated by workflows running on multiple data processing

R. Souza et al. *Efficient Runtime Capture of Multiworkflow Data Using Provenance*. Author preprint of paper accepted at the 15th IEEE International Conference on e-Science, San Diego, California, USA (eScience 2019).

systems. Users can run forward and backward provenance queries, integrating provenance traces generated in the multiworkflow. Despite its originality in addressing the limitations of single-workflow data capture, Komadu still suffers from capture overhead, which is particularly significant in CSE workflows, as shown in our experiments. In Komadu, performance issues related to runtime data capture are left for future work [8], since it integrates the provenance graphs for queries only after all workflow executions end. Finally, a limitation found in all related work [4], [6]–[12] is the lack of query support from workflow data stored in heterogeneous databases.

To mitigate these problems, we propose a system called ProvLake, which adopts design principles for providing efficient distributed domain data capture in a multiworkflow execution. While capturing the data, ProvLake logically integrates and ingests them into a provenance database, named ProvLake Data View (PLView), ready for analyses at runtime. We validated ProvLake by implementation of the real use case in Figure 1 with four workflows that process 5 TB datasets for a deep learning classifier. We evaluate runtime data analysis exploring heterogeneous multiworkflow data represented in our provenance database. We compare ProvLake and Komadu with extensive experiments and demonstrate ProvLake's overhead for runtime data capture to be negligible.

The main contributions of this paper are the following:

- Design principles for efficient distributed provenance data capture with low overhead.
- A provenance data representation aware of multiworkflows and multiple stores, following W3C PROV [5] standards.
- Lessons learned on efficiently keeping the overhead low while integrating captured provenance data.

The rest of this paper is organized as follows. Section II presents multiworkflow data representation as provenance data. Section III introduces the ProvLake system with its design principles for introducing low overhead in provenance data capture. Section IV has the experimental evaluation. Section V discusses related work and Section VI concludes.

II. Multiworkflow Data Provenance

A provenance database in a multiworkflow is the main source of runtime data analysis. Provenance data do not replicate data from the data stores. Instead, they contain lightweight references to the physical data residing in the data stores; strategic data values (*e.g.*, quantities of interest, performance indicators, or any other relevant value) extracted from the datasets in the multiple stores; and the data relationships among these data, providing the logical data integration, thus forming a data view over the multiworkflow data using provenance. PLView data only contain relatively small but relevant data that can be used for runtime analysis and to guide deeper analyses in the data store contents. Moreover, we adopt a strategy to promote the cooperation of the teams to decide on which data are relevant to them to set the provenance granularity. We present the fundamentals of the PLView and a methodology to select strategic data for analysis in Sections II.A and II.B, respectively.

A. PLView Provenance Data Representation

To relate distributed data from multiworkflow executions, including workflow data stored in heterogeneous databases, the PLView adopts data provenance relationships following a well stablished W3C standard among provenance data systems: PROV [5]. The PLView is represented as a provenance data directed graph, where vertices are instances of PROV entity, activity, or agent and edges are data relationships between vertices [5]. Despite PROV's high level representation, it can be specialized to represent workflow data relationships. The PLView represents data related through data transformations and relationships of data distributed in heterogeneous databases.

A workflow is a composition of data transformations (*e.g.*, programs, services, functions) that can consume and produce datasets, where an output dataset produced by a data transformation can be consumed as an input dataset by another data transformation, forming a coherent flow. A dataset can be modeled as a set of data elements, where each element is composed of data values. Each value has a data attribute, which gives the name and data type (*e.g.*, integer, string, array). Theses attribute names are typically specific for the domain and familiar to the teams. Data transformation executions are modeled as instances of activities and data values as instances of entities. The "consumption" data relationship between a data transformation execution and its data values is modeled as the *used* PROV relationship, whereas the "generation" of data values by a data transformation execution is modeled as the *generated* PROV relationship.

To improve data analysis, the PLView adds semantics for the attributes of the data values. The semantics refer to the meaning that the data value has in a data transformation. Possible values for attribute semantics are: a parameter or output value of a data transformation; data reference to a data element of a dataset physically residing in a data store; or data value extracted from a dataset in a data store. These fundamental concepts for workflow provenance are precisely defined in background work [3], which we base on to extend to represent data references (*i.e.*, data values that have attributes with semantics of data reference) to data relationships in heterogeneous databases.

To help the logical integration between data that are physically distributed into multiple stores, the PLView creates the *referred* to represent data relationships between data references. PLView goes one step further to relate data from heterogeneous databases. Examples of data references are file reference, document reference, relational tuple reference, graph vertex reference, RDF triple reference, etc. Therefore, the data reference is complemented via the *hadStore* relationship to relate it to its data store (also a PROV entity), that analogously can be a File System, Document DBMS, Relational DBMS, Graph DBMS, Triple Store, etc.

In addition, the PLView adds properties to vertices to improve runtime data analysis. For example, properties of data transformation executions are information about where they were physically executed, start time and end time and data references contain meaningful information about the data being referred, *e.g.*, size of files in case of file reference.

Altogether, the PLView is represented as a multiworkflow provenance data graph that provides a data view over the data in

the CSE project while the multiworkflow executes. We illustrate the PLView's provenance data representation in ProvLake's website [13] and a concrete example is presented in Section IV.

B. Methodology to Select Strategic Data for Analysis

To select strategic data for multiworkflow data analysis, we present a methodology that helps users decide which data are relevant and should be captured. This decision is made by the teams participating in the CSE project and is guided by the provenance questions that the teams want to answer at runtime. The granularity of captured data impacts both quality of its analysis and overhead for its acquisition. Considering this trade-off, we propose a methodology, which extends a single-workflow methodology [14], to drive the teams to design the data that will form the PLView. Using prospective provenance data representation [2], the methodology aims at specifying only the relevant data that should be captured and related as retrospective provenance data [2]. The methodology is analogous to modeling a relational schema, with the relationships between relations, in a relational DBMS.

The methodology phases are: (i) identification of data to be analyzed; (ii) specification of data capture points at workflow codes; and (iii) specification of attributes and relationships between data references. The phases are followed initially for each single-workflow, and then for the multiworkflow, globally. For phase (i), users anticipate interesting questions for data analysis within each workflow and the multiworkflow. Workflow modelers and data provenance specialist collaborate with the application developers, who are often computational scientists or engineers, and domain scientists to identify such questions, which will drive the identification of *strategic* data to be analyzed. By strategic we mean the input and output data values that are of high interest and should be captured. For phase (ii), the developers and workflow modelers identify, in the workflow code, all data transformations and their strategic input and output data anticipated in phase (i). The result of this phase drives the insertion of data capture calls in the corresponding workflow code. Finally, for phase (iii), they specify the semantics of data captured in the workflows, particularly attributes that are data references and how they are related to form the relationships between data references, similarly to what is done when designing join attributes in a relational schema.

All three phases are followed for each workflow, resulting in a design specification that represents the prospective provenance data of all data transformations for each workflow. Then, to form the multiworkflow provenance database, the teams collaborate, working in phases (i) and (ii), to specify (prospectively) the relationships between provenance data graphs of the single-workflows. For each pair of workflows, the teams decide on new attributes, the provenance data relationships between workflows, and data reference relationships between workflows for phase (iii). The result is the same prospective provenance specification, but with these added attributes and relationships.

The notion of "strategic data" may change over time. Thus, the methodology is iterative, and workflows' specifications can suffer adjustments during a CSE project's timeline as new data become of interest. Finally, after the multiworkflow specification

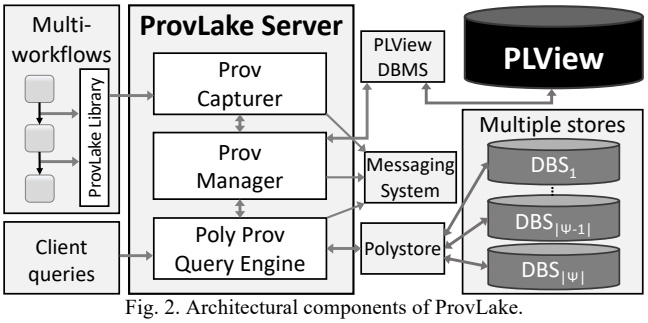

Fig. 2. Architectural components of ProvLake.

using prospective provenance data, ProvLake captures retrospective provenance data as the multiworkflow executes.

III. PROVLAKE ARCHITECTURE

This section presents ProvLake's architecture, beginning with an overview and design principles, then details of each component.

A. Overview and Design Principles

CSE users need to analyze multiworkflow data at runtime but cannot afford high computational overhead on their running workflows. Thus, ProvLake architectural design is focused on attaining low runtime data capture overhead.

ProvLake has a microservices architecture composed of three services (ProvCapturer, ProvManager, and PolyProvQueryEngine), a lightweight ProvLake library, a messaging system, and the PLView (Figure 2). These components capture data, transform them to the provenance database representation, insert them into the database, and help runtime data analysis through query submissions. We implement ProvLake following these main principles:

(i) Lightweight library. ProvLake provides a lightweight library to be imported into the workflows (clients), which adds little code for instrumentation; thus, avoiding significant pollution in the original workflows' code. The library only contains simple methods to capture input and output data values, exactly as they are in order to take advantage of cached data during capture (*in-situ*) and leave to the server the management of transformation of workflow data into provenance data, provenance-specific relationships, semantics, parallel insertions in the database, and other more heavyweight operations. Moreover, the server runs in a different address space of the running workflows, following *in-transit* strategies [15], contributing to avoid contention between clients and server. Additionally, to increase isolation between clients and server, ProvLake server components are suggested to be deployed on a separate hardware from where the workflows run. For instance, in an HPC cluster machine, ProvLake server runs on a node whereas the workflows run on the remainder nodes.

(ii) Asynchronicity. The communication between clients and server during execution of the data transformations is asynchronous. That is, requests are non-blocking and return almost instantaneously with simple "ACK" messages to clients.

(iii) Work queues. Clients do not communicate to server at each data capture, but they simply enqueue capture calls, which is a fast and local operation. When the queue reaches a certain limit or time constraint (both adjustable), the capture requests are

sent in a batch to the server. This reduces network traffic as the clients do fewer but larger requests.

In addition to these design principles, which are the main ones for keeping low overhead added to the workflows, other strategies collaborate to the overall server performance. The server services also employ queues and parallel workers to consume them. Also, the services maintain auxiliary in-memory data structures to store, for instance, the workflows' specifications, which data should be captured, and the data references to be related. Thus, ProvLake avoids reading data from disk, from the PLView, or any other external data store out of main memory during execution.

B. ProvLake library

ProvLake library follows the three design principles. They lower the overhead and the changed workflow code remains as close as possible to its original code. Figure 3 illustrates a small excerpt of Workflow 2 of our case study (Section IV.B), written in Python, with added library calls.

The library is imported (Line 1) in the code of each single-workflow composing the multiworkflow. As a result of the methodology (Section II.B), prospective provenance data of each single-workflow are specified in a separate configuration file, stored externally to the workflow code. This file is loaded only once, at the constructor of the library, to an in-memory data structure (Line 2). Then, each execution of a data transformation is wrapped by two data capture calls, one to capture the input (Line 9) and other for the output (Line 11) data values. These calls are queued, and the arguments of each data capture are often small lists and hash-tables, with data values in their original formats. Other calls or data conversions specifically related to provenance data are designed to remain separate from the workflow code and left to ProvLake server. The library also captures runtime information, such as start and end times of each data transformation execution, and information about the physical machine running the workflow. Figure 3 also exemplifies the library capturing a reference to data stored in heterogeneous data stores: the file system (by a file reference to a seismic data file in Line 4) and the MongoDB DBMS (by a reference to a document in Line 10).

C. ProvCapturer

ProvCapturer service has two main goals. First, to convert the workflow data coming from the library calls into W3C PROV data following PLView data representation and, second, to capture the data relationships. The service follows the prospective provenance data specification, loaded to its main memory.

As requests arrive, they are just appended to its in-memory queue, so the service can immediately "ACK" the message. This reduces waiting time in the workflows caused by the communication between the library and the service. This queue of requests is processed in parallel by the service.

To convert the data values coming from the workflows to the PLView data representation, the service matches the workflow data with the prospective provenance specification. Each data transformation execution call carries the identifiers of the data transformation and of the workflow with it. With this, ProvCapturer looks into the prospective provenance data to find the data attributes corresponding to the data values coming from the workflows, and how they should be converted to retrospective provenance data. The

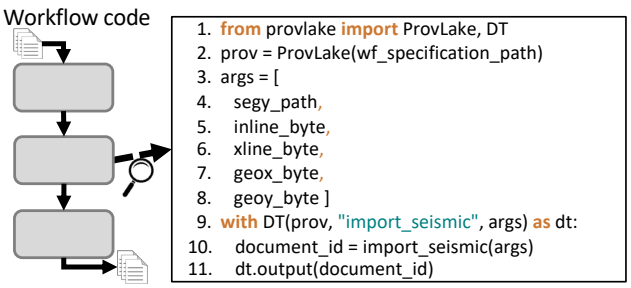

Fig. 3. Part of a workflow code with ProvLake library calls.

workflow data are transformed into JSON format following the W3C PROV-TEMPLATE specification extending the vocabulary utilized in the PLView data representation. These JSON objects are sent to the ProvManager service.

To capture the data relationships between data transformations and workflows, which are given by the consumed and generated data values, the service uses unique identifiers to every data value that flows through it. Using unique identifiers for maintaining relationships of captured data is used in several provenance systems [8,12]. Thus, every data value receives a unique identifier in the PLView. If a same data value that is generated by a data transformation execution is consumed by another, the service captures this and creates the data relationships that represent the shared data between these data transformations. Similarly, if this same data value generated in a workflow is consumed in another workflow, the service captures this, forming the data relationships between the provenance data graph of these workflows. To specify the unique identifier, the service uses a deterministic rule, which uses a hash function over the data value, the attribute, and CSE project identifier. When a new data transformation uses an already captured data value, the service gives the same identifier to it, creates the provenance relationships, and creates the JSON objects.

Special cases occur when the captured data values are data references. In those cases, the service creates the provenance relationship *hadStore* between the data reference and its physical data store. Information about the data store includes the data model and credentials for accessing the data store, if applicable. Another special case occurs when the data references participate in a relationship between data references. In this case, the service maintains the references in another in-memory data structure. When a data reference participating in a pair of data values that form the relationship flows into the service, the service checks if the other value in the pair has already been captured. If yes, it creates the provenance relationship *referred* between the data values. If not, it saves the value in the in-memory structure and the value will remain there until the other value in the pair flows into the service.

Furthermore, we design the service so that it does not make any assumption about execution dependencies or centralization of the multiworkflow execution. It is the service's responsibility to distinguish between the workflows sending data and to create the data relationships between provenance graphs of different workflows as the workflows execute. After processing the calls coming from the workflows, captured provenance data are sent via RESTful HTTP calls to ProvManager, also asynchronously.

D. ProvManager

ProvManager is responsible for inserting provenance data into the PLView's DBMS and for generating queries (in the query langue of PLView's DBMS) to be sent to the DBMS. When ProvManager receives the provenance data, it converts the data into a data format that can be inserted into the DBMS. In current implementation, the PLView Provenance Data Representation is instantiated as an ontology that extends W3C PROV-O, using AllegroGraph[1] as its Triple Store. Thus, ProvManager converts provenance data into RDF triples, and inserts them into AllegroGraph. ProvManager manages a queue of triples, to be inserted as a bulk to the DBMS, aiming at reducing contention at the DBMS. For queries, ProvManager receives calls from PolyProvQueryEngine service and builds the SPARQL queries to answer the calls.

E. PolyProvQueryEngine

To query PLView for multiworkflow data analysis through queries at runtime, ProvLake exposes a provenance query API via PolyProvQueryEngine, which implements parametrized predefined queries for multiworkflow provenance graph traversals and analytics. Users specify parameters, such as a source and target data attributes, to be traversed in the provenance graph stored in the PLView. PolyProvQueryEngine sends a query request to ProvManager only, which builds a SPARQL query to the DBMS and returns the result set. However, in certain cases, when data were not captured by ProvLake and still the user needs to query the data, with their provenance, PolyProvQueryEngine also sends a request to a Polystore and joins with the result set coming from a provenance query to ProvManager. Exploring the polystore queries aspect in depth is out of the scope of this paper.

F. Messaging System

Since the communication between components in ProvLake is done asynchronously during the data transformations and only return simple "ACK" messages, keeping track of their status is not trivial. "ACK" is not enough to determine whether the requests were completely processed. For this, we make use of a messaging system as a central log of status of the asynchronous requests. In current implementation, we use Apache Kafka. Each service publishes messages in its own channel to register the beginning and end of each processing of a request, and a status code and callback message (*e.g.*, "success" or a specific error message). In this way, users can check if their requests were fully processed (*i.e.*, sent to ProvCapturer, then to ProvManager, and finally inserted into the PLView) or an error occurred in a specific component.

IV. Experimental Evaluation

In this section, we provide an experimental evaluation of ProvLake. In Section IV.A, we present the analysis of data capture overhead using 36 synthetic workloads. In Section IV.B, we present a real case study, showing multiworkflow data analysis and overhead analysis. In Section IV.C we discuss lessons learned.

Hardware setup. All tests are conducted on a cluster of 12 machines, where each has 128GB RAM, two CPU Intel Xeon v2 2.8GHz with 20 cores when using hyper-threading, *i.e.*, 40 cores per machine summing 480 cores. They share GPFS with 24TB and are interconnected via an InfiniBand network.

Software setup. ProvLake services (ProvCapturer, ProvManager, and PolyProvQueryEngine) and its PLView DBMS are deployed on a Kubernetes[2] cluster of Docker containers on top of the physical cluster. The services are implemented using Python and deployed with uWSGI[3] with C++ Cython plugin with multi-process and multi-thread parallelism enabled. The DBMS is AllegroGraph 6.3. For Komadu deployment, we use the most up-to-date version available [17]. Komadu's services were compiled as indicated in its documentation. We also deploy Komadu on the same Kubernetes cluster.

A. Overhead Analysis

The experiments in this section aim at evaluating the overhead ProvLake adds to clients, *i.e.*, the workflows, under several synthetic workloads. To analyze overhead, we measure the execution of the workflows with and without data capturing enabled. Two dimensions are typically analyzed when evaluating scientific applications: *task duration* and *number of tasks* [18]. Since we are analyzing data capture overhead, we add a third dimension: number of captured *data values per task*. This quantity represents the amount of captured data for both input and output data values for each task; each data value is about same size. First, we present an overview of execution times and comparison with Komadu, then we discuss the overhead in detail.

Experiment setup. To vary in these three dimensions, we use a benchmark with synthetic workloads based on existing workflows, including for example the workflows in our case study, and on past work on scientific applications [18]. They mimic a prespecified number of chained data transformations, each processing multiple parallel tasks. There is a synchronization point between two chained data transformations, *i.e.*, before a new data transformation begins, all tasks of the current data transformation finish, which is a typical behavior in scientific applications. We generate quantities for the three analyzed dimensions following a normal distribution where the mean values are according to Table 1 and a standard deviation of 10.0. We use three chained data transformations in these experiments. In total, we generate 36

Table 1. Mean values to generate the synthetic workloads.

Number of tasks	–	30	300	3000
Task duration (s)	0.1	1	10	100
Data values per task	–	20	100	200

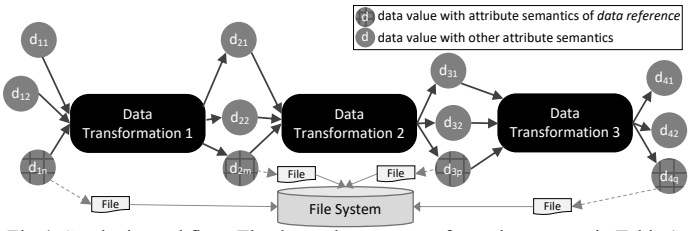

Fig 4. Synthetic workflow. The data values per transformation vary as in Table 1.

[1] https://franz.com/agraph/allegrograph/

[2] https://kubernetes.io

[3] https://uwsgi-docs.readthedocs.io/en/latest

workloads, which corresponds to the permutation of the values in Table 1. An illustration of this workflow is presented in Figure 4.

We increase the order of magnitude for each dimension to analyze the system under various workloads. An exception is number of data values because we do not know any realistic case that captures thousands of data values for one single task. For each data transformation, one of the input data values is a reference to a file in the file system. We use only one data store in this test to generate synthetic workloads that could be used within Komadu as well, and in ProvLake one data store is enough to test the overhead in the client-side. Also, for task duration, we use another case to investigate the system's performance for very short duration tasks (*e.g.*, each lasting for 0.1 seconds on average). Although tasks in scientific applications are often long-lasting [18], we produce workloads dominated by thousands of short-term tasks. This is a way to stress the system, which is one of the objectives of this experiment. Thus, for the largest case, there are about 3000 tasks (1000 parallel tasks on average per data transformation), each with a mean duration of 1.7 min (100 seconds), and for each task there is a mean amount of data values of 200 to be captured. To compare with Komadu, we implement an analogous version of the same workflow we use to test ProvLake. We add Komadu calls to capture data during execution of the workflow, similarly to what we do for ProvLake calls. We followed user guides and documentation publicly provided to fine tune configuration parameters, such as increasing queue sizes, so to better accommodate a high number of parallel tasks. Then, we test Komadu using the exact same 36 workloads of the synthetic workflow we use to test ProvLake.

Overview of execution times. Each of the 36 workloads is executed with the following three scenarios: (i) without any data capture, (ii) with ProvLake data capturers, and (iii) with Komadu data capturers. The total execution times to process the workloads are not deterministic and do not follow a normal distribution, thus we report the medians of a batch of repetitions. For each scenario, for each workload, we repeat at least 50 times and until the 95% confidence interval of the median is within 5% of our reported medians. Similarly, we do not plot error bars as they represent less than 5% of the medians. Results are in Figure 5. We organize the results using a 3x3 matrix, where each chart in the matrix is represented with a letter (A)—(I). The y-axis of each chart shows the Log Execution Time and we vary the mean task duration in the x-axis. In the matrix, by varying in the rows, we vary the log number of tasks. By varying in the columns, we vary the mean amount of captured data values per task.

Finding: execution times with ProvLake data capture remain close to the execution times without capture, in all 36 workloads. When the number of parallel tasks or data values per task increases, ProvLake runs significantly faster than Komadu.

Comparing with Komadu, in 10 workloads Komadu data capturers run at least one order of magnitude slower than with ProvLake's. For small number of tasks (A—C) and long-lasting tasks, both systems perform similarly. For workloads with hundreds of tasks (D—F), Komadu performs similarly to ProvLake only for 100 seconds of mean task duration. For all other cases, the difference is significant. The greatest difference occurs in chart (F), *i.e.*, hundreds of millisecond-sized tasks with 200 data values on average per task. While ProvLake executes 1.1x slower than without data capturers, Komadu executes 369x slower. Thus, ProvLake performs over two orders of magnitude faster than Komadu and in all tested cases ProvLake adds less overhead. We could not run the workloads

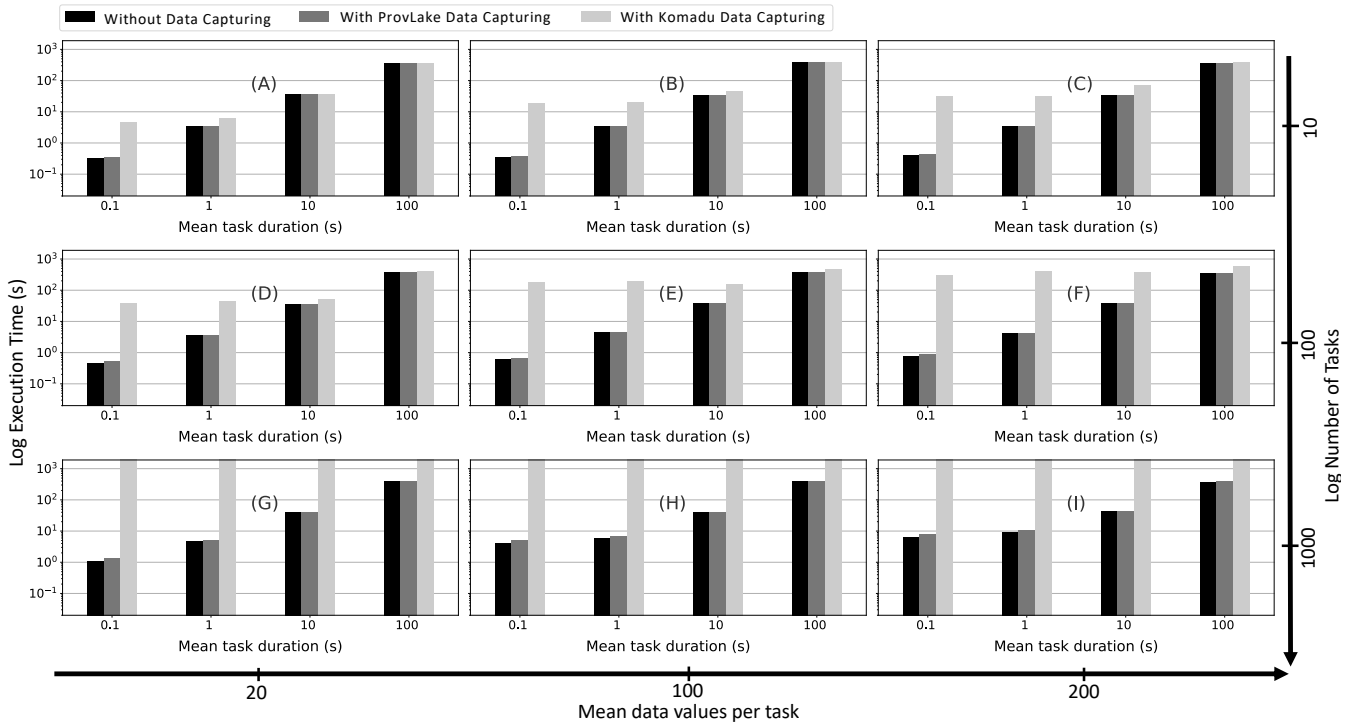

Fig. 5. Execution times with ProvLake, Komadu, and with no data capture on 36 workloads.

with thousands of parallel tasks (G—I) using Komadu because despite varying several settings following its user guides, Komadu throws timeout errors after thousands of parallel tasks are launched in our deployment.

Thus, we observe that the design principles adopted by ProvLake help to keep the overhead small. Komadu does not adopt the principle to provide a lightweight library, requiring the workflow code to be instrumented with W3C PROV activities, entities, and agents, and PROV relationships, embedded in the workflow code. In ProvLake, such PROV-specific modeling remains in the server (ProvCapturer) rather than in its client library. At runtime, it leads to more API, operating system, and service calls, increasing the competition between Komadu data capturers and the running workflow. Improving current instrumentation of the workflow code is planned as future work in Komadu [8].

Analyzing time overhead in more detail. Figure 6 shows the runtime data capture overhead of each of the 36 workloads. The percentages are obtained by measuring the relative difference between with and without ProvLake calls. Three charts are plotted, where each has a fixed number of tasks. In the x-axis, we vary the mean task duration and for each task duration, we plot three bars, each representing the amount of data values captured per task. The y-axis is the overhead percentage. We annotate the total execution time above the bars of the workloads without data capture for 200 mean data values per task.

Finding: the number of tasks has higher influence than the mean data values per task and the overhead decreases with task duration (i.e., total execution time).

An exception is for the millisecond-sized tasks workloads, as the overhead increases as the number of data values captured increases. This occurs because of ProvLake's initialization overhead, which is incurred for reading a workflow specification file from disk (prospective provenance) and populating in-memory data structures. When the workload has only tasks that execute in milliseconds, the execution time is dominated by this initialization time. This happens because with so many fast tasks, ProvLake queues get overloaded. Both the ProvCapturer API in the client-side takes longer to send requests and the ProvCapturer server takes longer to process all requests. However, these small workloads are useful to perform stress tests against the system, as realistic workloads usually last for several seconds or minutes.

For longer workloads, ProvLake's overhead is quite low. Even when capturing a larger amount of data values per task, it is not enough to significantly increase the execution time. When tasks last at least 10 seconds on average per task, the overhead is around 1%. For the workloads dominated by long-lasting tasks, as of 100 seconds on average, it adds about 0.1% of overhead, which is negligible.

B. Case Study and Multiworkflow Data Analysis

This section presents the case study that motivates this work. We start with an overview, then we describe the workflows, and queries which ProvLake can answer.

Discovery of oil reserves is paramount for the O&G industry and involves a broad spectrum of activities, including seismic image interpretation. Typically, these images cover large extents of the earth and by inspecting the images, geoscientists try to identify geological features, such as salt bodies. Trying to automate such activity is of high interest in both academia and O&G industry [19] and deep learning is a promising machine learning technique for this [20].

Managing the data lifecycle to train deep learning models is necessary to deliver models of high quality and this is particularly true in geoscience problems [21], such as identification of textures in seismic images [20]. It requires preprocessing, cleaning, and performing complex integrated data analysis. To deal with such complexity, the lifecycle is decomposed into parts, each addressed by different, collaborating teams of geoscientists, computational scientists, engineers, among others. Each team has a preferred way to automate tasks and store data, and a team consumes data generated by another. This case study focuses on activities that range from preprocessing large raw geological data files to the generation of training and validation datasets for deep learning models. Decomposing the problem into many workflows makes the problem feasible, however it creates a new problem: how to consume the data in an integrated way. Managing provenance in the data lifecycle in a well-structured manner becomes a major requirement as it facilitates the understanding of how models were generated and improves trust in the results.

The preprocessing part of the lifecycle is composed of four workflows (*c.f.* Figure 1). Workflow 1 processes about 5 TB of geological raw data files (mainly seismic files in SEG-Y format and intersecting horizons stored in CSV format). Despite its formal specification [22], SEG-Y files very often do not follow it, so lots of preprocessing and cleaning are needed. If erroneous raw data are used or if the data were not cleaned correctly, the generated training and validation datasets for the deep learning classifier, hence its results, cannot be trusted. To address this, Workflow 1 parses the files in the file system of the HPC cluster, extracts strategic data from the files, and automates data cleaning. Extracted data are stored in PostgreSQL.

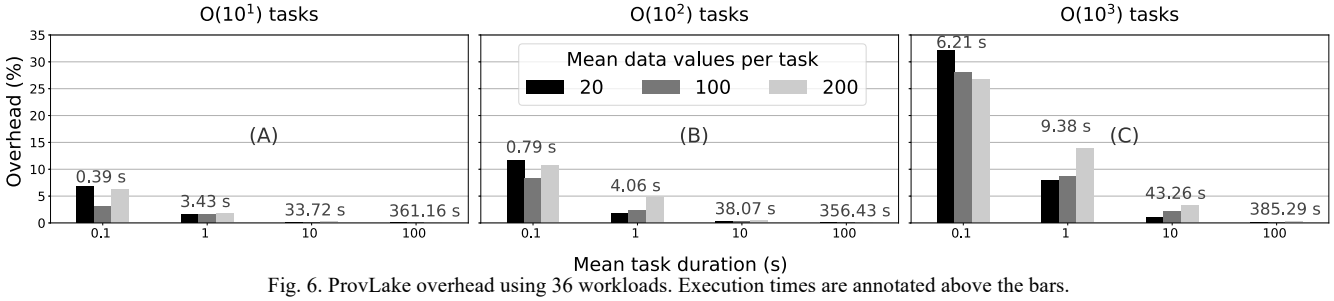

Fig. 6. ProvLake overhead using 36 workloads. Execution times are annotated above the bars.

Based on the outputs of Workflow 1, users inform which seismic and horizon files should proceed to the Workflows 2 and 3, respectively. These workflows generate intermediate data files and bounding boxes and populate a MongoDB database. Furthermore, knowledge and annotations about the seismic and horizons, known to geoscientists, are inserted in a knowledge graph database managed by the AllegroGraph DBMS. Finally, Workflow 4 consumes data from Workflows 2 and 3 to generate training and validation datasets, which are stored as files in a Kubernetes volume [23]. In the following steps of the data lifecycle, these files are used as input to train the models.

Multiworkflow Queries. Based on this case study, we identify distinct queries ProvLake was made to answer. In this analysis, the user is a computational scientist and a machine learning practitioner, with deep knowledge in the domain. When reporting results, she requires detailed information, such as which SEG-Y files, intermediate files, and documents in MongoDB were used to generate a set of training and validation files. Thus, the typical queries are multiworkflow data analysis:

Q1: *What were the data references, stored in datasets distributed over multiple stores, consumed and generated in the data generation process, throughout the workflows, of a given pair of training and validation files?*

Q2: *What are the relationships between the data references obtained in Q1?*

To provide detailed domain-specific information, such as which in-line and cross-line slices of the seismic cube, she explores the semantics of attributes of the data values (Section II.A) to inspect the inner contents of each data reference returned in Q1:

Q3: *List all data values (extracted from the files and datasets, parameters and output values of data transformations) to generate a given pair of training and validation files.*

In addition to comprehensive queries, she runs debugging queries. She observes that training and validation datasets that Workflow 4 is generating are producing models with unusual poor accuracy. She suspects that Workflow 1, which extracts strategic values (*e.g.*, geographic coordinates) from SEG-Y files, did not extract data correctly, and asks:

Q4: *How the geographic coordinates were extracted from the SEG-Y file that is being used to produce training and validation files? What is the spatial resolution between slices in the seismic data*?

Answering Q1—Q4. Figure 7 shows an excerpt of PLView's contents when the four workflows execute[4]. Rectangles with dashed strokes represent data store instances and the ones without dashed strokes are instances of data references. Also, all data references that are in a same data store follow the grayscale background color of the data store instance, illustrating the *hadStore* relationship (dashed arrow). The excerpt shows the data relationships when data containing a seismic cube acquired in Netherlands basin are processed in the four workflows. When ProvLake captures data values (like x, y coordinates of a seismic cube) extracted from datasets, it creates the corresponding relationships to the data references. Similarly, parameter values of a data transformation execution and data values that are output of the data transformation execution are related to the data transformation executions. These relationships of the data values that are not data references are not shown in the figure for the sake of its comprehension. We see the relationships between provenance data of different workflows and the relationships between data references in heterogeneous data stores in Figure 7. During an execution of a data transformation in Workflow 1, ProvLake registers that the data transformation *used* a seismic file (*netherlands.sgy*), extracted raw data from it, and *generated* a tuple in a relational table, which is stored in PostgreSQL. In addition, ProvLake creates relationships between *PostgreSQL1* and the data references, and stores the *referred* (solid arrow) relationship between the seismic file and the instance in the relational table. Similarly, as the other workflows execute, unique identifiers to every data value provide the relationships. Then, users send API calls to PolyProvQueryEngine.

Q1 and Q2. To illustrate, let us suppose that the user inputs of query Q1 are the training and validation files generated by Workflow 4 in Figure 7. In this case, the user sends an API call to return all data references related to the generation of training and validation TensorFlow records files, from raw seismic files to the TensorFlow records files. To return the results, PolyProvQueryEngine executes a query that takes training and validation file references (represented as files stored in the Kubernetes volume) as source nodes and traverses the data graph

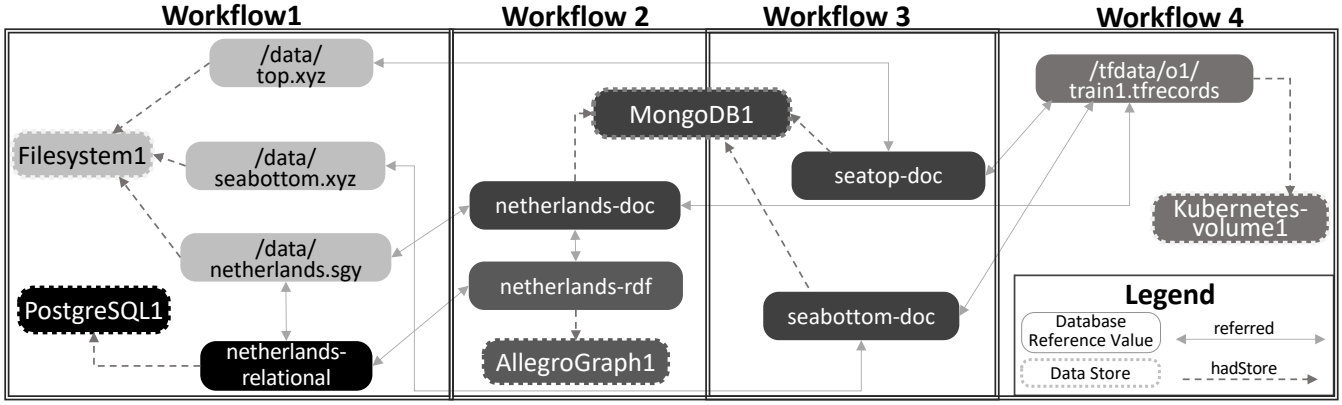

Fig. 7. Excerpt of PLView contents generated during the multiworkflow execution.

[4]A detailed version of this figure is available online [13].

backwards until the farthest data reference attribute values, which are seismic (*netherlands.sgy*) and horizon files (*seabottom.xyz* and *top.xyz*) and return the related references, via *referred*. Because the query is to answer all data references in the provenance data path, PolyProvQueryEngine uses SPARQL features (*e.g.*, property paths) to traverse the graph. The result set returns all data references displayed in Figure 7. For Q2, the result set is similar, but it shows the data references in Figure 7 and the edges of the graph between references.

Q3 and Q4. For Q3, the result shows the data references consumed and generated in all data transformations for all workflows and data values with other attribute semantics. The result of Q3 has the data needed by the user to compose a report on how the training and validation datasets for the deep learning classifier were generated from raw data, passing through all workflows. For Q4, the user specifies to show the values for data attributes *used* by the data transformation "SEG-Y data extraction" (*i.e.*, parameters of the transformation) and the x, y coordinates generated by it (*i.e.*, the extracted data from the SEG-Y data), and specifies to display the spatial resolution, which is a data attribute stored in MongoDB.

Then, the result is the geographic coordinates (*coordx, coordy*), extracted from the seismic file *netherlands.sgy* and stored in PostgreSQL in Workflow 1, along with parameters that were used to execute that data transformation that extracted the data. The parameters of this data transformation provide information about how the data extraction was run. Spatial resolution of seismic slices is stored in the document reference (generated in Workflow 2) which is related (*referred*) to the seismic file.

Overhead analysis. Most real workloads are composed of mixed tasks (short/long duration, few/many tasks and data values). This is the case of our real case study, where the characteristic of each task, hence the whole workload, can be mapped to the nearest workload tested in the previous experiments. For example, tasks in Workflow 1 last for up to few minutes and generate over 100 data values, depending on the size of the seismic file the task is processing. Some tasks in Workflows 2 and 3 are short (taking few seconds), whereas most of them last from few minutes to hours, depending on the seismic and horizon files being processed by a task. Each task in these workflows generates less than 30 data values. Tasks in Workflow 4 last several minutes, depending on the size of the region of the seismic cube being processed. Each task generates about 50 data values to be captured. Thus, the majority of tasks of the workloads are mapped to long duration tasks with dozens to hundreds of data values. Few tasks are mapped to short duration tasks and very few last for less than one second. Considering space overhead, Workflow 1 produces about 5.3 million provenance tuples in the PLView's DBMS, Workflows 2 and 3 together produce about 400,000 tuples, and Workflow 4 produces about 100,000 tuples. In total, the DBMS storage used about 4.5 GB, whereas the workflows processed over 5 TB of geological data files. Therefore, *for the real workflows, ProvLake adds about 0.1% of time overhead to each workflow individually and generates about 5.8 M tuples and 4.5 GB in the PLView's DBMS.*

In addition to the experiments presented, we also investigated other aspects of ProvLake, such as different work queue sizes observed that when sizes are set to one, PLView has high frequent data insertions, which increases overhead, but provides near real time data available for runtime queries. Due to space limitations, we provide examples and further implementation details in ProvLake's website [13].

C. Lessons Learned

For runtime data capture, in all presented workloads, ProvLake has shown a predictable behavior and managed to maintain low data capture overhead. Particularly, when workloads are dominated by tasks that last for 10 seconds or more, ProvLake's overhead is negligible, even for a large number of parallel tasks and captured data values. Since CSE workloads are typically dominated by long-lasting tasks that last more than a minute each [18], ProvLake is a good solution for this class of applications. When comparing with Komadu, we see that when the amount of parallel tasks and amount of data values per task are small, both systems perform similarly. Nevertheless, when the workload scale grows, as is the case with CSE workflows, the design principles adopted by ProvLake keep the overhead low. A major difference in both systems' design is that providing a lightweight library is a design principle in ProvLake. Aligning this with asynchronous calls and work queues with parallel processing were critical design decisions that contributed to achieving the results. As a result, the design principles adopted by ProvLake allowed for having an overhead two orders of magnitude lower than Komadu in CSE workflows.

For the runtime analytical queries, the methodology to design the data in the PLView promoted cooperation among the multidisciplinary teams so they could specify relationships, and which data should be captured at runtime, driving success to answer the multiworkflow queries. ProvLake's ontology, with its adherence to W3C PROV [5], also contributed to the queries.

V. Related Work

We organize the related work according to addressed issues: (i) runtime data analysis with low workflow overhead and (ii) capturing data relationships from multiworkflow data. We group related work as: runtime single-workflow data capture; multiworkflow orchestration systems; integration of heterogeneous provenance databases; and runtime multiworkflow provenance data capture.

Runtime single-workflow data capture. Solutions in this group capture data of workflows at runtime and store as provenance data [4], [9]. For instance, DfAnalyzer [4] captures implicit relationships between data files, associating them to data extracted from files. This raw data extraction is convenient for analyzing related domain data directly from its provenance database. DfAnalyzer has low data capture overhead in large-scale CSE workflows and influenced ProvLake's design principles. However, solutions in this group are limited to data analysis of single isolated workflows. They do not address the issue that workflows run autonomously but implicitly sharing data. Captured data are specific to a single workflow, without explicit interconnections between workflows. Also, these

solutions disregard that multiworkflow data analysis often requires data integration of data in multiple stores.

Runtime multiworkflow provenance data capture. Komadu [8,12] is the only solution we found in this group. Different from the previous groups, Komadu aims at generating integrated provenance data as a multiworkflow runs. Users add data capturers to existing workflows to collect and relate data that flow in the workflows. Then, Komadu allows for forward and backward provenance queries, and joining provenance traces in the multiworkflow. It can integrate provenance of data generated in data lakes and by workflow orchestration systems, like Spark or WMSs. However, its data representation, hence data capturing capabilities, disregard that data are often processed in multiple stores, jeopardizing the multiworkflow analysis. Additionally, regarding the overhead, the authors report significant overhead added to the running workflows. In the experimental evaluation of this paper, we show that ProvLake outperforms Komadu in a wide variety of workloads.

Multiworkflow orchestration systems. QoX [10] and PAW [11] aim at optimizing the execution of multiworkflows that process data in a variety of parallel execution engines (*e.g.*, Hadoop MapReduce and Spark) and use multiple stores (e.g., HDFS, NoSQL, and relational DBMSs). SHIWA [24] provides efficient execution management in a multiworkflow environment, focusing on scalable mechanisms for orchestrating workflows in single Workflow Management Systems (WMSs). However, WMSs and parallel execution engines are often not adopted by CSE users, who frequently adopt libraries with their own parallel execution control, which conflicts with a workflow scheduling engine [14]. In addition, none of these solutions provide an integrated view over the multiworkflow.

Integration of heterogeneous provenance databases. Solutions in this group [6,7] aim at interoperability in heterogeneous provenance databases. This is useful in multiworkflow environments where each workflow engine generates provenance data using its own specific format. However, the drawbacks of these solutions are that they also provide provenance data integration offline.

To summarize, we did not find any solution that copes with the two issues addressed by our solution and combining existing approaches into one is hard. It requires new concepts for multiworkflow provenance data, a practical methodology for multiworkflow data design, and design principles for runtime multiworkflow data capture with low overhead.

VI. Conclusion

In this paper, we introduced ProvLake, a system that addresses the challenge of efficient multiworkflow provenance data capture with low overhead. By capturing strategic data values and their data relationships, ProvLake maintains the PLView to provide a logical integration of multiworkflow data at runtime.

We proposed a specialization of a provenance data representation, which stem from relationships between data references stored in distributed and heterogeneous data stores and provenance graphs of different workflows. We followed W3C PROV to design an ontological data representation for the PLView. To enable the instantiation of the PLView to multiworkflows, we organized a set of phases in a methodology to specify which data values should be captured driven by the relevant queries. Finally, we proposed design principles that contributed to providing the runtime analysis, as evidenced in our real case study, while keeping the overhead as low as 0.1%. Compared with Komadu [8,24], the closest solution that meets our goals, our approach enabled runtime multiworkflow data analysis with much smaller overhead.

VII. Acknowledgments

The authors would like to thank Marcelo Costalonga, Lucas Villa Real, Rodrigo Ferreira, Daniel Salles, Daniela Szwarcman, Maximilien de Bayser, Viviane Torres, and Marcio Moreno from IBM Research for their help during the development of this work. This work was partially funded by CNPq, FAPERJ, and Inria Associated Team SciDISC.

VIII. References

[1] M. Atkinson, S. Gesing, J. Montagnat, and I. Taylor, "Scientific workflows: past, present and future," *FGCS*, vol. 75, pp. 216–227, 2017.

[2] M. Herschel, R. Diestelkämper, and H. Ben Lahmar, "A survey on provenance: What for? What form? What from?," *VLDB J.*, vol. 26, no. 6, pp. 881–906, 2017.

[3] R. Souza, V. Silva, J. J. Camata, A. L. G. A. Coutinho, P. Valduriez, and M. Mattoso, "Keeping track of user steering actions in dynamic workflows," *FGCS*, vol. 99, pp. 624–643, 2019.

[4] V. Silva, D. de Oliveira, P. Valduriez, and M. Mattoso, "DfAnalyzer: runtime dataflow analysis of scientific applications using provenance," *PVLDB*, vol. 11, no. 12, pp. 2082–2085, 2018.

[5] P. Groth and L. Moreau, "W3C PROV: an overview of the PROV family of documents," 2013. https://www.w3.org/TR/prov-overview/.

[6] A. Gaignard, K. Belhajjame, and H. Skaf-Molli, "SHARP: harmonizing and bridging cross-workflow provenance," in *The Semantic Web: ESWC 2017 Satellite Events*, 2017, pp. 219–234.

[7] P. Missier *et al.*, "Linking multiple workflow provenance traces for interoperable collaborative science," in *WORKS*, 2010.

[8] I. Suriarachchi and B. Plale, "Crossing analytics systems: a case for integrated provenance in data lakes," *IEEE eScience*, pp. 349–354, 2016.

[9] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "noWorkflow: a tool for collecting, analyzing, and managing provenance from Python scripts," *PVLDB*, vol. 10, no. 12, pp. 1841–1844, 2017.

[10] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal, "Optimizing analytic data flows for multiple execution engines," in *SIGMOD*, 2012.

[11] K. Doka *et al.*, "Optimizing, planning and executing analytics workflows over multiple engines," in *EDBT/ICDT Workshops*, 2016.

[12] I. Suriarachchi, S. Withana, and B. Plale, "Big provenance stream processing for data intensive computations," in *IEEE eScience*, 2018.

[13] "ProvLake website," 2019. https://ibm.biz/provlake.

[14] V. Silva, R. Souza, J. Camata, A. L. G. A. Coutinho, P. Valduriez, and M. Mattoso, "Capturing provenance for runtime data analysis in computational science and engineering applications," in *IPAW*, 2018, pp. 183–187.

[15] A. C. Bauer *et al.*, "In situ methods, infrastructures, and applications on high performance computing platforms," *Computer Graphics Forum*, vol. 35, no. 3, pp. 577–597, 2016.

[16] L. Bavoil *et al.*, "VisTrails: enabling interactive multiple-view visualizations," in *IEEE Visualization*, 2005, pp. 135–142.

[17] "Komadu website," https://pti.iu.edu/impact/data-sets/komadu.html.

[18] I. Raicu, I. T. Foster, and Y. Zhao, "Many-Task Computing for Grids and Supercomputers," in *MTAGS*, 2008.

[19] T. Randen *et al.*, "Three-dimensional texture attributes for seismic data analysis," in *SEG Technical Program Expanded Abstracts*, 2000.

[20] D. S. Chevitarese, D. Szwarcman, E. V. Brazil, and B. Zadrozny, "Efficient classification of seismic textures," in *IJCNN*, 2018.

[21] Y. Gil *et al.*, "Intelligent systems for geosciences: an essential research agenda," *CACM*, vol. 62, no. 1, pp. 76–84, 2018.

[22] K. Barry, D. Cavers, and C. Kneale, "Recommended standards for digital tape formats," *Geophysics*, vol. 40, no. 2, pp. 344–352, 1975.

[23] "Kubernetes Volumes," 2019. https://kubernetes.io/docs/concepts/storage/volumes.

[24] D. Rogers *et al.*, "Bundle and Pool Architecture for Multi-Language, Robust, Scalable Workflow Executions," *J. Grid Comp.*, 2013.