A Survey on Collecting, Managing, and Analyzing Provenance from Scripts

JOÃO FELIPE PIMENTEL, Universidade Federal Fluminense, Brazil
JULIANA FREIRE, New York University, United States of America
LEONARDO MURTA and VANESSA BRAGANHOLO, Universidade Federal Fluminense

Scripts are widely used to design and run scientific experiments. Scripting languages are easy to learn and use, and they allow complex tasks to be specified and executed in fewer steps than with traditional programming languages. However, they also have important limitations for reproducibility and data management. As experiments are iteratively refined, it is challenging to reason about each experiment run (or trial), to keep track of the association between trials and experiment instances as well as the differences across trials, and to connect results to specific input data and parameters. Approaches have been proposed that address these limitations by collecting, managing, and analyzing the provenance of scripts. In this article, we survey the state of the art in provenance for scripts. We have identified the approaches by following an exhaustive protocol of forward and backward literature snowballing. Based on a detailed study, we propose a taxonomy and classify the approaches using this taxonomy.

CCS Concepts: • **Information systems** → **Data provenance**; • **Software and its engineering** → *Scripting languages*;

Additional Key Words and Phrases: Provenance, scripts, collecting, managing, analyzing, survey

ACM Reference format:
João Felipe Pimentel, Juliana Freire, Leonardo Murta, and Vanessa Braganholo. 2019. A Survey on Collecting, Managing, and Analyzing Provenance from Scripts. *ACM Comput. Surv.* 52, 3, Article 47 (June 2019), 38 pages.
https://doi.org/10.1145/3311955

1 INTRODUCTION

Computing and data have revolutionized science and enabled many important discoveries. At the same time, the large volumes of data being manipulated, the complex computational process used, and the ability to run experiments at a high rate create new challenges for reasoning about results as well as managing the data and computations.

This work is supported by CAPES, CNPq, FAPERJ, Moore-Sloan Data Science Environment at NYU, AT&T, DARPA, and NSF. J. F. Pimentel is a CAPES scholarship holder (PDSE/Process No. 88881.131563/2016-01). J. Freire is partially funded by the Moore-Sloan Data Science Environment at NYU, AT&T, DARPA, NSF Awards No. CNS-1229185, No. CNS-1405927, and No. CCF-1533564.

Authors' addresses: J. F. Pimentel, Universidade Federal Fluminense, Av. Gal. Milton Tavares de Souza, s/n. Niterói, RJ, Brazil; email: jpimentel@ic.uff.br; J. Freire, New York University, 2 MetroTech Center, Brooklyn, New York, NY, United States of America; email: juliana.freire@nyu.edu; L. Murta, Universidade Federal Fluminense; email: leomurta@ic.uff.br; V. Braganholo, Universidade Federal Fluminense; email: vanessa@ic.uff.br.

0360-0300/2019/06-ART47 $15.00
https://doi.org/10.1145/3311955

Systematic mechanisms to collect provenance for computational experiments are critical to address these challenges. Provenance refers to the documented history of processes in the life cycle of a computational object [95]. In the context of scientific experiments, provenance considers input and output data, environment characteristics, processes applied to input data to derive output data, intermediate data of these processes, and execution attributes such as duration of each process and of the experiment itself. Provenance enables scientists to reason about results. For example, to assess how many trial-and-error paths produced a particular result, how a given result was derived, and which processes led to a given result [37]. Provenance has many other applications. Scientists can use provenance to share experiment results with computation and input data [66], allowing others to reproduce them [21], check integrity and authenticity [81], and track the evolution of the experiments [106]. Additionally, scientists can analyze provenance to assess data quality, audit, understand experiments, and detect system dependencies [27, 97].

Scientific Workflow Management Systems (SWfMS) [14, 79, 129, 132] assist users in composing, executing, and collecting provenance from experiments. These systems glue components as execution plans in the form of workflows, which are essentially directed acyclic graphs (DAG) representing computations [19]. During the execution of these components, SWfMS can transparently collect their provenance. Despite their ability to define experiments and extensive support for provenance, a broader adoption of SWfMS has been hampered due to their steep learning curve and high adoption costs, since they require external tools to be wrapped into the workflow engine [98]. Some SWfMS, such as Swift [132], Snakemake [72], and dispel4py [36], proposed scripting languages for defining workflows but restrict the language to a syntax that supports the creation of a DAG, and thus they lack the flexibility provided by general-purpose scripting languages.

The power of general purpose scripts in gluing components and dealing with heterogeneous, combined with ease of use, were key factors in their wide adoption by the scientific community. Dubois [33] advocates using scripting languages such as Python, Perl, Matlab, and so on, for scientific programming instead of compiled programs. He claims these languages incorporate sophisticated data structures and give immediate feedback on algorithms. Similarly, Langtangen [73] attributes the growth of script usage in scientific experiments in part due to their simple syntax and ability to easily visualize results and combine different tools. Jackson [64] states the importance of Python for applications in science and engineering due to its simplicity, extensive built-in library, dynamic typing with support for object-oriented paradigm, and support for integrating externally compiled code, among other reasons. Finally, some initiatives (e.g., Software Carpentry[1]) use scripts for teaching computing skills to scientists.

Compared to SWfMS, one drawback of scripts is the lack of support for provenance collection. Recognizing this limitation, several approaches have been proposed to collect, manage, and analyze provenance from scripts. Each one of these approaches proposes different mechanisms for collecting, managing, and analyzing different types of provenance in scrips with multiple goals. In this work, we propose a classification taxonomy for approaches that work with provenance from scripts, and we classify the existing state-of-the-art approaches according to this taxonomy.

Multiple surveys have been written about provenance. Some characterize data provenance in e-Science [50, 118], provenance in computational tasks in general [37], provenance in databases [123], data-intensive scientific workflow management [80], and provenance in the light of Big Data [127]. Others focus on more specific aspects, such as dynamic steering [84] and provenance analytics [101]. However, none of them consider provenance from scripts. In this article,

[1]http://software-carpentry.org/.

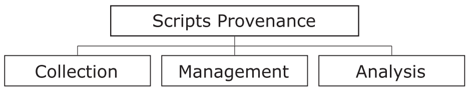

Fig. 1. Main taxonomy of provenance from scripts.

we aim to fill this gap by providing a comprehensive survey of existing techniques that address different problems related to provenance for scripts. As we describe below, we have created a comprehensive list of techniques through forward and backward literature snowballing. We hope that our survey and taxonomy will serve not only to organize the existing knowledge on provenance for scripts but also as a guide to help scientists to select tools that best address their specific problems.

The remainder of this article is organized as follows. Section 2 presents the fundamental problems and techniques for collecting, managing, and analyzing provenance from scripts. We present different types of provenance and discuss techniques for collecting, storing, and versioning provenance; methods that use provenance for reproducibility; and approaches for querying and visualizing provenance. Section 3 presents a systematic mapping of approaches that collect provenance from scripts. We classify the collection, management, and analysis techniques of each approach according to the proposed taxonomy. Finally, Section 4 concludes this survey presenting our findings and open research opportunities.

2 A TAXONOMY FOR PROVENANCE FROM SCRIPTS

In this section, we start by giving a brief overview of related work on capturing provenance for binary and source code. We then present a taxonomy for script provenance, which, as illustrated in Figure 1, considers techniques for collecting (Section 2.1), managing (Section 2.2), and analyzing provenance (Section 2.3). We also discuss the applicability of this taxonomy to other provenance systems (e.g., SWfMS and Database) and their differences to scripts (Section 2.4).

Tools for Collecting Provenance for Binary and Source Code. Many approaches have been proposed to collect provenance from binary executions (e.g., PASS [97], ReproZip [21], CDE [56], DataTracker [120], and others). They collect information about operating system processes, system calls, file objects, and network packets as provenance. Since scripts run in binary interpreters, these approaches can also be used to collect provenance for the execution of scripts. However, as they do not take the structure of scripts into account, it can be challenging to link the provenance they collect back to the steps in the script.

Besides using provenance tools, some benefits of provenance for scripts (e.g., reproducibility and comprehension) can be achieved by other tools. Version control systems can store, version, and distribute experiment definitions through repositories. For simple experiments that do not use environment information nor external tools, this may be sufficient for reproducibility, and for managing multiple executions. For more complex experiments, virtual machines can provide isolated environments and improve their reproducibility. While these tools allow scientists to reproduce experiments, they neither connect output to input nor help users to understand the experiments. Nevertheless, the literate programming paradigm [69] may help understanding experiments by encouraging users to describe what their code does. This paradigm encourages the writing of documents that combine, human-readable code descriptions, and computation results. However, this paradigm does not guarantee the reproducibility, since it does not keep track of the environment and input data. Some tools that use scripting languages and support literate programming, such as Jupyter [115] may also benefit from additional provenance collected from scripts [109].

```
1  import numpy as np
2  from provtool import where
3  # Precipitation input from Rio de Janeiro
4  input_file = where("p13.dat", "BDMEP-Rio-2013")
5  year = 2013
6  # Classification
7  data = np.genfromtxt(input_file, delimiter=";")
8  total = sum(data[:,3])  # provenance: skip-details
9  classification = "above" if total > 1172.9 else "below"
10 # classification.csv is generated from multiple executions of
11 # this experiment with different inputs. It depends on the input_file
12 with open("classification.csv", "a") as file:
13     file.write("{},{},{}\n".format(year, total, classification))
```

Fig. 2. Toy experiment that classifies a yearly precipitation data from Rio de Janeiro.

2.1 Provenance Collection

Provenance can be described according to different aspects and each aspect requires different collection mechanisms. Over the past two decades, some classifications for provenance have been proposed for describing such mechanisms. Before discussing the collection techniques in scripts, we use Figure 2 as an example to compare the previously proposed classification systems and establish one for this document. This example presents a toy experiment that classifies the yearly precipitation data from Rio de Janeiro as above average or below average. Note that we use this example to discuss not only its definition but also its trials. A trial is one execution of an experiment.

Cheney et al. [20] classify provenance in *why*, *how*, and *where*. Why-provenance identifies the data that were transformed into a new data object. The why-provenance of “classification.csv” in Figure 2 includes “classification” in line 9, “total” in line 8, “year” in line 5, and the file “p13.dat” in line 7 (variable “input_file”). How-provenance identifies the process (i.e., all the transformations that occurred). In Figure 2, the how-provenance includes the “np.genfromtxt” in line 7, “sum” in line 8, the if expression in line 9, and “format” in line 13. Where-provenance identifies the location from which the data object was extracted. Figure 2 identifies that “p13.dat” was obtained from BDMEP [2] in line 4.

While this classification system is relevant for database provenance, it may not be appropriate for scripts. First, the separation between why-provenance and how-provenance is not always clear. The number “1172.9” in line 9 of Figure 2 could be perceived either as why-provenance, as it is the data that determines whether the result will be “above” or “below,” or perceived as how-provenance, as it determines how to classify the data. Second, most scripts do not indicate the where-provenance of data. One could classify files locations as where-provenance, however, the file location is also encoded in the why-provenance of variables. Finally, this classification system lacks other types of provenance, related to the structural and environment information of the experiment.

The most common classification for computational tasks distinguishes provenance as *prospective* and *retrospective* [77, 134]. Retrospective provenance combines why-provenance and how-provenance to provide an understanding of the execution process, identifying what really happened during the execution. However, prospective provenance refers to the structure of the experiment (workflow, script, input files), and what is necessary to reproduce it (dependencies, environment). While the prospective provenance of Figure 2 includes the script itself and the modules

[2]BDMEP is a meteorological database for teaching and research.

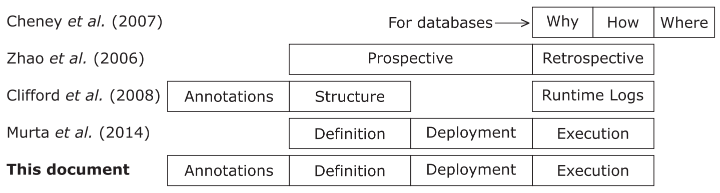

Fig. 3. Provenance classification systems.

"numpy" and "provtool," the retrospective provenance includes the execution flow and the parts of the script that were executed. In this case, the retrospective provenance indicates that the value of "classification" is "above." For the purpose of this survey, this classification system encodes too much information in the prospective provenance and lacks a different type of provenance.

Clifford et al. [24] propose a similar classification with three categories: *program structure*, *runtime logs*, and *annotations*. In this system, runtime logs correspond to retrospective provenance and program structure corresponds to the structural part of the prospective provenance. This system does not consider environment information. The third category in this system, annotations, refers to user-made annotations in the provenance or structure, which allow users to explain the program. In Figure 2, lines 10 and 11 present a provenance annotation in the form of a commentary that describes the origin of "classification.csv." Moreover, the "where" function call in line 4 is also an annotation, as it does not influence program execution and describes the origin of "p13.dat."

Murta et al. [98] borrows terms from software engineering [126] and classifies provenance for scripts in three categories: *definition*, *deployment*, and *execution*. Definition provenance represents the structure of the experiment, such as scripts and input files. Thus, it is equivalent to the program structure category proposed by Clifford et al. [24]. In Figure 2, definition provenance represents the script itself and "p13.dat." Deployment provenance represents the execution environment, with information about the operating system, dependencies, and environment variables. In Figure 2, deployment provenance represents the modules "numpy" and "provtool." Definition provenance together with deployment provenance corresponds to prospective provenance. Finally, execution provenance corresponds to runtime logs and retrospective provenance.

Figure 3 presents the aforementioned classification systems for provenance. Note that for the remaining of this document, we use the classification proposed by Murta et al. [98] due to its explicit separation of definition and deployment provenance, together with the *annotations* provenance proposed by Clifford et al. [24].

Each provenance type requires different collection mechanisms. While collecting annotations requires a way to parse annotations, collecting deployment provenance requires obtaining environment information with a completely different mechanism. However, collection mechanisms are not restricted to a single provenance type. Some mechanisms combine different provenance types. For instance, it is possible to use annotations to identify when and how to collect execution provenance [8, 86]. In this section, we present different collection mechanisms for each provenance type. Figure 4 presents the collection taxonomy.

2.1.1 *Annotations.*

According to Clifford et al. [24], users can make annotations either on procedures or on data. Additionally, we identify that some approaches also support annotations on provenance itself [27]. Annotations provide additional information about objects and users can use them to point interesting things, understand datasets and programs, and enrich data or provenance with more information [27]. Additionally, annotations can facilitate collecting other provenance types [8, 75, 86]. We classify annotations in five axes as presented in Figure 4: placement, extraction, target, inclusiveness, and necessity.

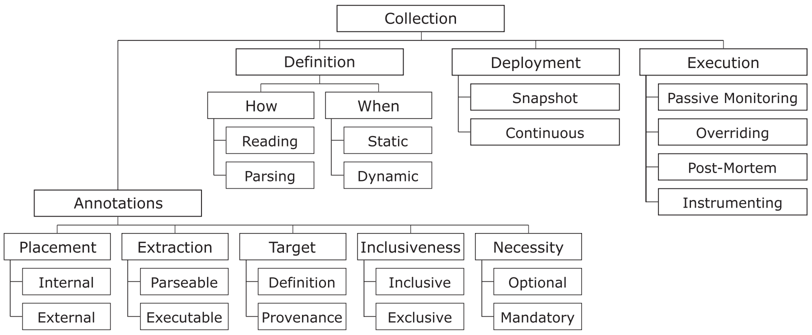

Fig. 4. Expanded *Collection* taxonomy node of Figure 1.

The placement axis classifies annotations according to their placement as *internal* or *external*. Internal annotations occur inside scripts or data and require some sort of extraction. That is the case of the annotations that appear in Figure 2. However, external annotations occur outside scripts and require a system that supports identifying data elements, through URI, provenance queries, or temporal information (e.g., annotating the last produced provenance).

The extraction axis classifies annotations according to their extraction mode as *parseable* or *executable*. Provenance systems can extract parseable annotations statically. However, executable annotations require their execution. In Figure 2, "where" in line 4 is an executable annotation, as it is necessary to execute it to get its result. However, the commentaries on lines 10 and 11 are parseable.

The inclusiveness axis classifies annotations as *inclusive* or *exclusive*. Inclusive annotations point things of interest and enrich data with more information. Exclusive annotations filter out uninteresting data or provenance. The annotations in Figure 2 are inclusive, but the commentary annotation in line 8 is exclusive, as it indicates that the details of the line are not relevant.

The target axis classifies annotations according to what they describe. Annotations can describe the program *definition*, including data and structure, or enrich the *provenance* itself. All annotations in Figure 2 describe the program. An example of annotation on provenance would be a tag on the trial indicating what it did.

Finally, the necessity axis classifies provenance according to the requirement of using them. Annotations can be either *mandatory* or *optional* for the systems that collect them. If the provenance system relies on annotations to collect provenance, then the annotations are mandatory. Otherwise, if it only uses annotations to enrich or filter the provenance collection, annotations are optional.

2.1.2 *Definition Provenance.*

Definition provenance refers to the project structure with scripts and input data. Collecting definition provenance can be as coarse-grained as collecting whole files [26, 27] or as fine-grained as extracting structure information from scripts to describe them [62, 87, 98].

The easiest way to collect coarse-grained definition provenance is to collect whole files as the definition of experiments. In this sense, version control systems [35] can help with definition provenance collection [27]. Besides the script and input file content collection, version control systems also provide authorship, creation timestamp, and script evolution as metadata for files. Instead of using version control systems, it is also possible to collect whole files during execution by applying execution provenance strategies as we discuss in Section 2.1.4 and collecting the files as soon as the execution tries to access it [26, 98]. This is especially valid for scripts, since interpreters read their definition before running them. However, this strategy may generate only a partial definition provenance of the project according to the execution path [56].

For finer-grained collection, it is necessary to statically analyze the structure [62]. Due to the unpredictability of dynamic languages [130], performing static analysis over scripts may not be enough to describe them. An alternative to cope with this challenge is to use annotations to describe the structure [8, 87]. However, this alternative is error-prone and may not represent the script definition. Using static analysis without user input reduces the possibility of errors but also limits the extraction of relevant information.

We classify definition provenance according to *how* and *when* it is collected, as presented in Figure 4. Definition provenance can be collected by *reading* whole files or *parsing* files and extracting information from them. In Figure 2, if we collect the whole script file, we will have definition provenance by reading. However, if we parse the file and extract information from it, then we will have definition provenance by parsing. Additionally, definition provenance can be collected *statically*, before or after the execution, or *dynamically*, during the execution. In Figure 2, it is possible to collect the script definition statically, before the execution, and the definition of "p13.dat" dynamically, when the program executes line 7.

2.1.3 Deployment Provenance. Deployment provenance represents the execution environment. It refers to the operating system version, interpreter version, environment variables, dependencies to programs and modules, and all the remaining deployment information that describes the environment. Most deployment information, such as operating system version, interpreter version, and machine specification, does not change during execution. Thus, it is safe to collect a single snapshot of such information. However, other deployment information may not be available at a given time for a snapshot or may change during execution. This is the case for module and program dependencies and environment variables. Hence, the strategies we describe in Section 2.1.4 for execution provenance also apply for continuously collecting such deployment provenance during execution [21]. However, since this information rarely changes during execution and some scripting environments support discovering dependencies without executing the script (e.g., Python's `modulefinder` discovers all imported modules), it is often worth to collect deployment provenance once, in a snapshot [27, 98] to avoid the overhead of dynamic provenance collection [19]. As presented in Figure 4, we classify deployment provenance according to its collection frequency, as *snapshot* or *continuous*. In Figure 2, we could collect the modules "numpy" and "provtool" as deployment provenance continuously during the execution of lines 1 and 2, respectively, or we could parse the script, extract the `import` information and collect a snapshot of the modules.

2.1.4 Execution Provenance. Execution provenance refers to the origin of data and its derivation process during execution. Different approaches collect both data provenance and process provenance at different granularities. Data objects can range from memory bytes to system objects, passing through arguments, variables, and network packets. However, the process can range from individual data operations to operating system processes, passing through variables operations and function calls. Due to the benefits of keeping the data for analysis and reproducibility [71], some collection mechanisms presented in this section support collecting not only metadata but also data itself.

Even though execution provenance appears in different granularities, it is possible to collect all granularities with similar strategies. According to Frew et al. [43], there are three strategies for collecting execution provenance: passive monitoring, overriding, and instrumentation. The passive monitoring strategy traces the process execution to collect provenance without requiring any modifications to the code. The overriding strategy replaces portions of the executed code with instrumented versions. Finally, the instrumentation strategy requires users to instrument their

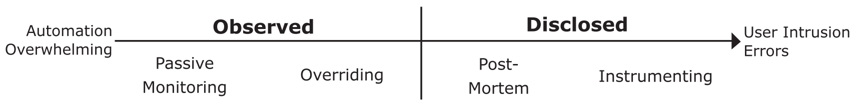

Fig. 5. Observed and disclosed strategies.

code explicitly with annotations or function calls. We identify a fourth strategy: post-mortem, which infers execution provenance after the execution [27, 61, 86].

Each one of these strategies has advantages and disadvantages. Passive monitoring and overriding are highly automated strategies but produce too much provenance, which affects the performance and overwhelms users. Instrumentation and post-mortem, however, require users to specify what they want to collect, being error-prone and producing less provenance. Braun et al. [13] separate provenance systems into observed and disclosed. Systems that apply passive monitoring or overriding are observed systems, since they observe the execution and collect provenance. Systems that apply post-mortem or instrumentation strategies are disclosed systems, since the users need to specify what they want to collect with annotations. Figure 5 presents an axis with all strategies. In the axis, the higher the automation, the more overwhelming its provenance will be. Note that the post-mortem strategy requires more automation than instrumentations. It occurs because post-mortem systems automatically infer provenance from results instead of having to specify each provenance collection.

The passive monitoring strategy uses a *tracer* to observe the execution and log all low-level events during the execution. Since tracers log all low-level events, this strategy imposes the biggest performance overhead, but it is also able to collect more provenance data. For scripts, it is either possible to trace interpreters' binaries [56] or to use language-specific tracers to collect provenance [8, 98]. This survey focuses on the latter. In Figure 2, the passive monitoring could trace all executed lines and collect the provenance in each one of them.

The overriding strategy automatically instruments the code to collect provenance. Provenance tools that employ this strategy define code patterns to find (e.g., function calls, file openings, variable assignments, and others) in the interpreter's binary or script and replace the original code with an instrumented one that collects provenance. In Figure 2, the overriding strategy could replace the functions that open files (e.g., "genfromtxt" and "open") by instrumented versions that collect provenance.

After overriding the code or tracing events, it is desirable to build a provenance DAG, which allows answering lineage queries. It can be accomplished by observing simple relationships, such as caller-callee function and parent-child process, and observing input and output data in each process. Another way to build a provenance DAG is to use a more robust technique such as dynamic program slicing or dynamic taint tracking to follow the actual data derivations that occur during executions. While the former approaches produce more false positives (i.e., find "provenance" that does not influence the results), the latter approaches produce more false negatives (i.e., do not find all the provenance that could influence the results). This occurs because dynamic program slicing and dynamic taint tracking just observe what occurred and not what could occur in other conditions [56]. Note that these robust techniques are also more expensive due to the necessity of following all dependencies at fine-grain.

The post-mortem strategy infers provenance from execution results after the executions. To collect this type of provenance, users need to specify the locations of output data and how it relates to input data. One way to apply the post-mortem strategy is to store all data files in a specific directory and collect all files before and after the execution. This method considers new or changed files as output files and unchanged files as input files [27]. Alternatively, it is possible

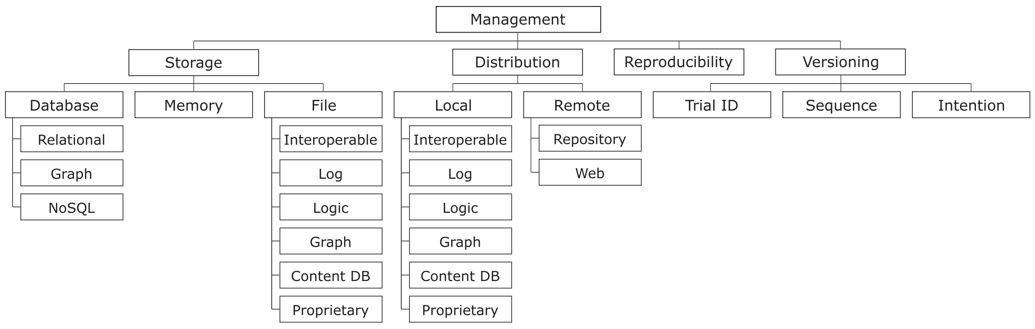

Fig. 6. Expanded *Management* taxonomy node of Figure 1.

to read all files in a directory after the execution and infer file provenance (i.e., which file derived from which files) through semantic similarities and timing information [61]. Another way to apply the post-mortem strategy is to use annotations [86] to collect the relationship between input data and output files. In both cases, users need to change their scripts to comply with the post-mortem rules, by using only the data directory or the annotation syntax.

The post-mortem strategy can also be joined to other strategies to collect provenance. For instance, it is possible to track process openings with the overriding strategy and collect files before and after each process execution, comparing them with the post-mortem strategy [1]. In Figure 2, the post-mortem strategy could be used to collect the resulting "classification.csv" after the script execution and associate it with the input file "p13.dat." Note that this strategy could also be used to collect implicit provenance (i.e., provenance data that is not explicitly referenced by the script [83]). In Figure 2, suppose the "where" function in line 4 extracts and reads "p13.dat" from a zip file, "precipitation.zip." The post-mortem strategy would be able to collect it and indicate that "classication.csv" derives from it.

Finally, the instrumentation strategy requires users to change their code specifying what they want to collect. Users can either annotate their code with special structures, such as decorators [8] or invoke library functions [12, 40]. This strategy not only imposes an extra effort for users but can also result in instrumentations that do not represent the scripts after code maintenance or due to human error [8, 87]. For this reason, PrIMe [91] has been proposed as a methodology for analyzing applications and determining which points should be instrumented, minimizing errors. Alternatively, the instrumentation strategy can also be used together with the aforementioned overriding and passive monitoring strategies to specify when to start collecting provenance and how to enrich the collected provenance [75]. In Figure 2, the "where" call in line 4 is an application of the instrumentation strategy.

2.2 Provenance Management

Collecting provenance data is not enough for provenance tools. It is desirable to provide management features related to *storage*, *distribution*, *versioning*, and *reproducibility*. In this section, we present provenance management requirements and approaches. Figure 6 presents the management taxonomy.

2.2.1 Storage. Provenance can be stored in *database* systems, transient *memory*, or *files*. The storage choice deeply relates to provenance collection and usage goals. File systems (e.g., archives, version control systems) are usually employed for reproducibility and definition provenance storage [27]. However, database systems work better for provenance comprehension and for storing other types of provenance due to the possibility of querying and the capability of storing non-file artifacts, such as function calls, variables, and environment variables [62, 98]. Although file systems are also viable for such non-file data, they require the provenance tools to implement their own serialization mechanisms [54, 103, 121].

Storing files in file systems and archives is straightforward. It just requires copying files from original paths to adjusted ones inside the storage system. However, since some scripts write in the same files more than once during its execution, it is often desirable to avoid collisions and collect more than one version of each file. One way to accomplish this is to define naming rules based on hashes of files content, and store files in a *content database*. In this case, part of the hash is used to define the name of the directory and another part to define the filename, with an external index to relate the original file name and version to its hash [27, 54, 98]. It is necessary to split the hash into different parts for directories and filenames to avoid OS limitations on the number of files that can be stored in a directory [98]. Such collision avoidance approaches are not necessary, should the collection keep only the most recent versions [90].

As mentioned before, database systems have advantages over file systems for supporting non-file artifacts and supporting queries. The chosen database system for each provenance tool also varies according to the necessities. Tools that intend to support simple queries use embedded *relational* databases such as SQLite [27, 62, 98]. However, due to the necessity of transitive closure queries and the unintuitive support for recursive queries in SQL, some of these tools also support exporting provenance to other formats, such as Prolog/Datalog [98]. This necessity of transitive closures also motivated some tools to use *graph* databases and other *NoSQL* databases right away [12, 17, 43, 51, 82].

The different nature of provenance artifacts indicates the need for combining different storage systems into a single tool. For instance, it is possible to store actual files in the disk or version control system and their relationships in a relational database [27, 98].

Using a storage system for provenance is not mandatory. Provenance tools can store provenance in a small set of documents, such as RDF, XML, JSON, Prolog/Datalog, non-structured log, among others [8, 43, 75, 87, 90, 121]. Other tools (or the same tool) might open these documents for analysis [75, 120], reproducibility [121], or querying [87]. Additionally, provenance might not be stored at all, should the application consume it at run-time [117]. In this case, provenance stays in transient memory. Moreover, instead of providing a storage system, an approach might output provenance in the standard output or distribute it through remote network connections and expect other applications to deal with the storage [12, 124].

2.2.2 Distribution. Besides storing provenance data, another provenance management issue is on distributing provenance to other people or systems for analysis and reproducibility. Distributing provenance for analysis allows tools to implement standalone collection mechanisms [124] and transfer the analysis responsibility to specialized tools. Distributing provenance for reproducibility reduces the burdens of making computation experiments reproducible across platforms [21].

Provenance tools that store provenance at a small set of *files* [8, 43, 75, 87, 90, 121] support distribution by simply sending the files to someone else. Other tools need to process provenance data and produce the desirable file format [98]. However, the desirable file format depends on its application. Logic programming formats (e.g., Prolog and Datalog files) support running queries with transitive closures [87, 98]. Graph formats (e.g., GraphViz files) allow visual analysis [1, 105]. Provenance-specific formats (e.g., OPM and PROV files) support interoperability among provenance tools and usage of other tools specialized in provenance querying and visualization [90]. Finally, it is also possible to distribute provenance as executable logs [90], which are representations of experiments without loops, conditions, and other control flows.

The Open Provenance Model (OPM) was proposed as the result of Provenance Challenges with the goals of supporting digital provenance representation of anything, with coexisting multiple levels of description, and a format that could be exchanged among systems [94]. The OPM specification heavily influenced the W3C PROV standard [96]. Both models are extensible and provide similar concepts and relationships for entities, activities, and agents. The relationships indicate whether an activity used or generated an entity; whether an entity derived another entity; whether an activity was associated with an agent; among others [25, 92].

All these formats provide distributable provenance but do not deal with the problem of provenance transferring. Thus, we define them as *local* distribution. RDFa [5] supports embedding some of these formats (e.g., PROV) in web pages. A user interested in embedded provenance can use RDFa parser to extract it. However, not all distributable provenance can be embedded. To support provenance transferring, some approaches propose sending the provenance to *remote* servers. These servers appear both as *web servers* designed to receive and store provenance data [12, 52] and as *repositories* designed to share provenance and experiment definitions, encouraging the reuse of experiments of other people [66]. Version control system repositories [35] play a similar role in distributing experiments. However, they usually only distribute script definitions and they make it hard to search for other types of provenance. Conversely systems provide versioning for the experiments.

2.2.3 *Reproducibility.*

Reproducible research is essential for science. In the scientific method, scientists confirm or refute hypotheses based on testable and reproducible predictions. The lack of reproducibility prevents other scientists to validate research findings and expand its horizons with new data [9]. With the advance of computers, the amount of data used in research got bigger, and it became unfeasible to reproduce research just with the data reported in papers [31]. This situation leads to a credibility crisis [63].

In response to the credibility crisis, scientists proposed sharing not only findings but also data, programs, and environments [23], making data as transparent and available as possible [58]. Provenance comes to play in these proposals due to its capability of representing data, data processing with intermediate transformations, and environment information.

Scientists can use provenance to comprehend third-party experiments and reproduce behaviors in new implementations and even compare different executions to check if a new trial could replicate the results of the previous one [27, 57].

According to Drummond [32], just replicating experiments results is not good science, as it just reports the same result originally reported and is only able to detect frauds. However, replicating experiments could be an important step toward reproducibility, since it allows scientists to check whether they are using the same proposed data transformations and tools before trying new data.

In this document, we do not propose a classification for reproducibility, thus we consider all approaches that aim at supporting replication, reproduction, or repetition of experiments as tools that support reproducibility.

2.2.4 *Versioning.*

Many experiment results motivate repetitions in their life cycle [85]. For instance, when a trial is inconclusive, scientists may repeat the cycle to adapt hypotheses and tasks. When scientists confirm a hypothesis for a restricted population, they may repeat the experiment for a broader one. Similarly, when they refute a hypothesis for a broad population, they may verify it for a restricted one. Moreover, some scientists design experiments to run iteratively, alternating the input data and some experimental activities. For instance, this occurs in simulations with parameter sweeping. In these simulations, each iteration deals with a combination of input parameters. In all the situations that motivate repetition, the knowledge is cumulative and scientists can use data from previous trials in further analyses. Some experiments may even use the output of

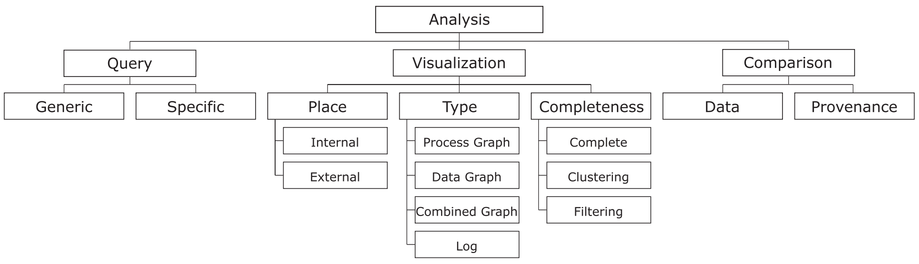

Fig. 7. Expanded *Analysis* taxonomy node of Figure 1.

a trial (i.e., one execution of an experiment [106]) as another trial's input. Finally, some scientist may desire to rollback to previous versions of the experiment with interesting results.

While collection mechanisms presented in Section 2.1 collect provenance of a single trial, these mechanisms leave the experiment evolution out. However, as the experiment evolves, its trial provenance evolves as well. Thus, to keep all trial provenance, it is necessary to version it for different executions.

In its essence, versioning provenance requires just to provide a way for separating provenance storage for each execution. Using a *trial identification* for collected provenance [98, 121] is sufficient to identify each execution. Ideally, such systems should apply optimizations to reduce storage overhead and facilitate analyses.

However, just specifying trial versions is not enough to understand the evolution. Suppose that a trial uses a file created by a previous trial as input. In this situation, the provenance tool should consider the provenance of the file in the previous trial for the new trial. Having just unordered versions does not allow one to identify which version was the previous one. Thus, in addition to versions, it is necessary to track provenance evolution in the form of version relationships [27, 106].

Trial relationships represent how the experiment evolves by indicating situations such as sequential trial executions or re-executing previous trial versions. This way, they improve provenance across trials and, consequently, help during analysis. Hence, provenance evolution allows users to not only analyze the latest script provenance but also to compare it to previous moments and improve their understanding of the whole experiment. Note that the trial relationships can be as simple as the *trial sequence* [98], or as complete as indicating the *evolution intention* [106].

While provenance evolution has been applied to SWfMS [14], it has not received much attention for scripts. A possible reason is the wide usage of version control systems to track the evolution of script definitions [35], which fills part of the necessity of evolution tracking. Note that provenance tools that use version control systems for storage also support trial provenance evolution tracking [27, 121].

2.3 Provenance Analysis

Provenance analysis aims at supporting the comprehension of data and processes. Analyzing provenance involves visualizing and querying provenance data. Provenance visualizations provide an overview of what happened in a trial and what data derivations occurred. Provenance queries obtain lineage and other metadata from data objects. This section presents different approaches for querying, visualizing and comparing provenance. Figure 7 presents the analysis taxonomy.

2.3.1 Query. Many approaches use *generic* languages for querying provenance, such as SQL [27, 98], SPARQL [18, 82], XQuery [12], Prolog [98], and Datalog [87, 133]. Even though these logic programming languages (i.e., Prolog and Datalog) are not proper query languages, deductive databases use these languages as query languages due to their increased power in comparison to conventional SQL [111]. In the context of provenance, this increased power helps with recursive queries and transitive closures. While SQL supports recursive queries with transitive closures, those queries are known to be inefficient and hard to write [98]. Logic programming languages intuitively handle recursion, however.

Generic query languages are useful to users who know their syntax but can be complicated to deal with structured provenance data [37]. Additionally, the lack of knowledge about the internal storage structure increases the difficulty of provenance utilization. Thus, some *specific* query languages have been proposed for provenance, such as OPQL [78], VQuel [16], and other proprietary ones for specific systems [75].

OPQL [78] was designed to run specialized queries on provenance modeled with the Open Provenance Model (OPM). Its queries combine basic set operations (union, insert, and minus) and graph navigation constructs that support exploring transitive closures or single edges of OPM.

VQuel [16] was proposed as a generalization of the Quel [122] language with features of GEM [131] and path-based query languages. It has the goals of traversing version-level provenance information, querying data contained in a version, and comparing it to other versions. While VQuel focuses on the provenance of versions, it can also be used to query provenance evolution, should the content of each version be trial provenance.

While most existing querying languages focuses on offline analysis (i.e., after execution), provenance querying can safely occur online (i.e., during execution) to obtain derivations up to a determined moment [84]. Querying online provenance externally helps to identify problems as soon as possible in long-running programs and stop the execution before waiting a long time for their completion [25]. Querying online provenance internally (i.e., by the program that is producing it) improves the usage of intermediate data. Intermediate provenance data allows caching results and identifying differences between executions to invalidate caches [54].

2.3.2 Visualization. As we mentioned before, some approaches export provenance as interoperable files (e.g., OPM, PROV) for visualization in *external* tools [90, 124]. However, since provenance can be tight to a domain or not exported to interoperable files, some approaches that collect provenance offer their own *internal* visualization mechanisms [1, 34, 62, 70, 75, 87, 98].

Most approaches visualize provenance either as *logs* [51] or as directed *graphs* [1, 34, 62, 70, 75, 87, 98]. Such graphs present data transformations, data communication between activities, or activities sequence. Different graph views can represent the same provenance information according to the analysis goal [87]. *Data-centric* views present data as nodes and activities that apply transformations over data as edges. *Process-centric* views present activities as nodes and data transference between activities as edges. Finally, *combined views* present both data and activities as nodes and their relationships as edges. Combined views often include authorship as well [94].

Some *complete* provenance graphs are overwhelmingly big. Thus, it is necessary to summarize provenance through *clustering* or *filtering* to support visualization analysis in such graphs. Provenance clustering combines similar nodes and edges in the provenance graph. It can be performed manually [34, 62] or automatically [70, 98]. Manual approaches require users to select which nodes they want to combine into a single node. Automatic approaches use similarity measures for clustering. The similarity measures might consider provenance sequencing [70] or not [98]. Approaches that do not consider sequencing can break acyclic constraints of provenance during summarizations. These constraints can be purposely broken to represent script cycles in visualizations [98]. Dynamic visualization tools can represent clusters as collapsible nodes [75].

For provenance filtering, it is possible to use query languages described in Section 2.3.1. Some query languages are distributed with provenance browsers that support provenance visualization [6]. Alternatively, it is possible to filter provenance with simple predefined filters, such as temporal filters for selecting provenance data produced in a specific time range [70].

Graphs are not the only way to visualize provenance. Sankey Diagrams are an alternative that supports visualizing the magnitude of flows in activities network [59]. Visualizing the magnitude of flows helps to determine important activities based on dataflow. Among the existing approaches that support provenance visualization, some are coupled with the infrastructure that collects the provenance [1, 62, 75, 87, 98] and others intend to be generic for any provenance application [59, 70]. Generic approaches use interoperable provenance formats (e.g., OPM, PROV), as discussed in Section 2.2.2. They have the advantage of supporting provenance from different sources. Coupled approaches read provenance directly from the provenance storage system. They have the advantages of considering collection characteristics and improving visualization semantics.

2.3.3 Comparison. Some provenance approaches support comparing *data* to present differences between results [27] and for cache invalidation [55]. Others support comparing *provenance* graphs to understand differences between executions [10, 38]. Since comparing general graphs is equivalent to the sub-graph isomorphism problem, which is NP-complete [119], some approaches reduce the complexity of the comparison by using the system context. The system context can indicate the lack of loops in graphs [38], the guarantee of well-formed loops for trials written in SPFL (series-parallel graph overlaid with well-nested forking and looping) [10], and other information that is specific to each provenance system.

2.4 Applicability to Other Provenance Systems

We designed the proposed taxonomy for scripts, but some of the described features also apply to other approaches that collect, manage, or analyze provenance in non-scripting languages [22, 52], binary program executions [21, 28], operating systems [57, 97], scientific workflow management systems [38, 79, 129], and database systems [20]. In this section, we contrast these systems to scripts and compare the applicability of the taxonomy.

Usually, **Non-scripting Languages** (also known as system programming languages) are more verbose, with variable declarations, data and code segregation, and well-defined substructures, procedures, and components [102]. Provenance collection in these languages benefits from more informative static program analysis techniques than scripts [22]. For instance, since components are known in advance, it is easier to collect libraries as a *deployment* provenance *snapshot*, during the compilation. Similarly, *parsing* the source code to collect the *definition* provenance before the execution provides more information on types and dependencies than scripts provide. This information can be used to ease the *execution* provenance collection by *overriding* fewer parts of the program. In contrast, scripts are less verbose and designed for gluing distinct components with non-informative interfaces. Thus, scripts require more dynamic effort in the provenance collection.

When collecting provenance from **Binary Program Executions**, the program is dissociated from the source code definition [21, 28]. On the one hand, it allows users to collect provenance from any executable. On the other hand, it hinders the understanding and limits the provenance collection. For instance, *annotations* can only occur *externally*, since the collection does not have access to the source code for extracting *internal* annotations. As a consequence, the *instrumentation* strategy cannot be used for binary *execution* provenance collection. Additionally, the *definition* provenance collection cannot rely on *parsing* the source code. Thus, binary approaches use the *reading* strategy to collect input/output files and executable files.

Operating Systems provenance is very similar to binary provenance and all binary restrictions apply. Approaches of this category collect provenance of everything that is running in the operating system. Thus, associating the execution provenance to source code definitions is even harder. Moreover, since the collection occurs during the OS execution, both the *definition* and the *deployment* provenance are collected *dynamically* and *continuously* during the execution. Operating systems also imposes challenges on provenance *storage* due to the presence of the database on the operating system. Hence, the system must avoid collecting provenance of it to avoid recursive provenance. Additionally, the provenance of all processes imposes scalability issues on the *storage* and *analysis*.

Scientific Workflow Management Systems collect workflow activities as *definition* provenance by *statically parsing* the workflow structure [38, 79, 129]. It allows their *annotations* to *target* only the *provenance* instead of the *definition*. Since SWfMS define their own execution machinery, they do not employ the *overriding* strategy nor the *instrumentation* strategy for *execution* provenance collection. Instead, they use only the *passive monitoring* strategy for explicit provenance and the *post-mortem* strategy for implicit provenance.

An important distinction between SWfMS and scripts is the granularity of collection. Ordinarily, SWfMS collect only activities and data passing between activities. Most of the time, these activities are black-box operations and the SMfMS must assume that activities outputs derive from all the inputs. In scripts, activities can be expressions evaluations, function calls, and even script executions. Scripts express not only these activities invocations but also their definitions. This allows scripts to treat activities as white-box operations and obtain more precision. Note, however, that not all activities are white-box operations in scripts. Calls to compiled or built-in functions are black-box operations. Additionally, some SWfMS support sub-activities [129], and some approaches propose combining SWfMS to external tools to fill the black-boxes [15] (e.g., using a scripting approach to collect provenance from a workflow activity that invokes a script).

Another distinction between SWfMS and scripts is the mutability of the data [110]. Scripts can have mutable complex data structures. The mutability imposes an additional challenge in the collection. Suppose two activities apparently receive the same data structure, but only one of them performs changes in the data. In this case, the order in which the activities are executed influences the results. Additionally, nested data structures in scripts hinder the understanding of the provenance and require more advanced collection strategies.

Database Systems have three types of provenance: why, how, and where [20]. Our taxonomy does not model *where-provenance*, as this information is very rare in non-database systems and appear as part of other provenance types in scripts (see the discussion in Section 2.1). Additionally, we combine both *why-provenance* and *how-provenance* into the *execution* provenance, since it is harder to dissociate these concepts on scripts. Usually, database systems do not collect *definition* nor *deployment* provenance, since they are interested in the provenance of the stored data. *Annotations* are *parseable* and *target* the provenance. Thus, database systems do not use the *instrumentation* strategy for *why* and *how* provenance collection. Naturally, database systems use their own storage for provenance, but some approaches also support exporting it to other formats. Finally, *versioning* is different in these systems, since the concept of trial does not apply for database systems.

3 STATE-OF-THE-ART TOOLS ON PROVENANCE FROM SCRIPTS

We conducted a systematic mapping to identify the state-of-the-art tools on provenance from scripts. According to Petersen et al. [104], the main goals of a systematic mapping are producing an overview of a research area, categorize existing work, and explore tendencies. In our case, the systematic mapping has the goal of identifying tools that deal with provenance from scripts and categorize them according to their goals, and how they perform provenance collection, analysis, and management. Thus, we defined the main research question and five secondary questions:

- **RQ1:** Which provenance tools deal with provenance from scripts?
- **RQ1.1:** For what purpose do these tools collect provenance?
- **RQ1.2:** Where and when were these tools published?
- **RQ1.3:** How do these tools collect provenance?
- **RQ1.4:** How do these tools manage provenance?
- **RQ1.5:** How do these tools analyze provenance?

We applied forward and backward snowballing to discover relevant tools [128]. The snowballing method starts with a start set of papers related to the systematic mapping research questions. Forward snowballing consists in obtaining papers that cite papers in the current set and including them in the set if they match the inclusion criteria. Similarly, backward snowballing consists in obtaining papers in the references list of papers in the current set and including them in the set if they match the inclusion criteria.

In our case, we defined the **inclusion criteria** as peer-reviewed documents (e.g., papers, theses) in English with approaches that collect, manage, or analyze provenance from scripts directly. We **excluded** approaches with indirect support for provenance (e.g., virtual machines for deployment provenance) and approaches for provenance in non-scripting languages (e.g., Java [52]), generic binary executables (e.g., ReproZip [21], DataTracker [120]), or OS (e.g., PASS [97], Burrito [57]). While binary and OS-based approaches support collecting script provenance by monitoring interpreters, we left them out because of their dissociation between script definition and execution.

We followed the guidelines proposed by Wohlin [128] for defining the start set of our snowballing (i.e., use Google Scholar to avoid bias toward a publisher; and obtain a diverse and big enough start set). We searched "script provenance" on Google Scholar, and we selected papers based on our inclusion criteria. We obtained nine papers [29, 43, 44, 62, 75, 82, 86, 87, 98] related to seven approaches, and we stopped on page 5 after the page did not contribute new results. These papers were published in two distinct journals and three distinct conferences.

Then, we exhaustively alternated series of backward and forward snowballing iterations with the help of a snowballing tool (https://joaofelipe.github.io/snowballing/) until no more related papers were obtained. We finished the process on March 6, 2017. Figure 8 presents the process and the amount of related and found papers in each step. Note that this figure does not represent the actual process, but summarizes it satisfactorily. The actual process was performed over several months with many intermediary forward snowballing steps. For instance, the first forward snowballing on July 24, 2016, found only 24 papers that cited the first noWorkflow paper [98], according to Google Scholar. In the latest iteration, there were 34 citations for this paper. Thus, instead of presenting the whole snowballing process in Figure 8, we present only what it would be if we had performed the whole snowballing on March 6, 2017, with big backward and forward iterations, as described by Wohlin [128]. Note that the last two iterations were applied over the *s4* set, as they did not include related papers. During this process, we visited 1,345 references and we ended up with 53 papers referring to 27 approaches. In the remaining of this section, we describe all these approaches. Figure 9 presents the work we selected in the snowballing. The full citation graph with the reasons some work do not match the inclusion criteria is available at https://dew-uff.github.io/scripts-provenance/.

After selecting the papers and classifying the approaches they describe according to the taxonomy described in Section 2, we contacted the authors of each approach to confirm the classification. We received answers from authors of 19 approaches. This feedback made us realize that some papers are part of bigger systems [46–49, 93]. Additionally, some authors indicated the inclusion of newer papers of their approaches [67, 76, 108].

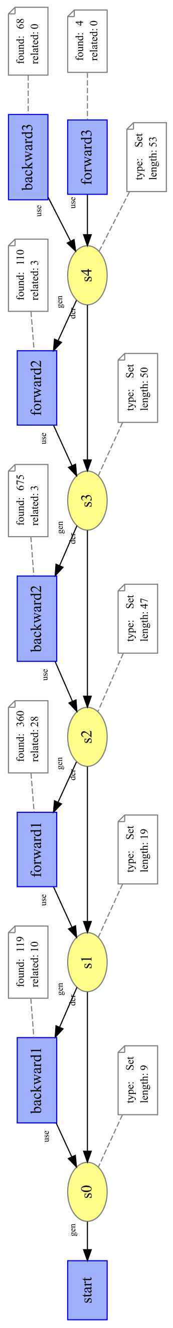

Fig. 8. Snowballing provenance.

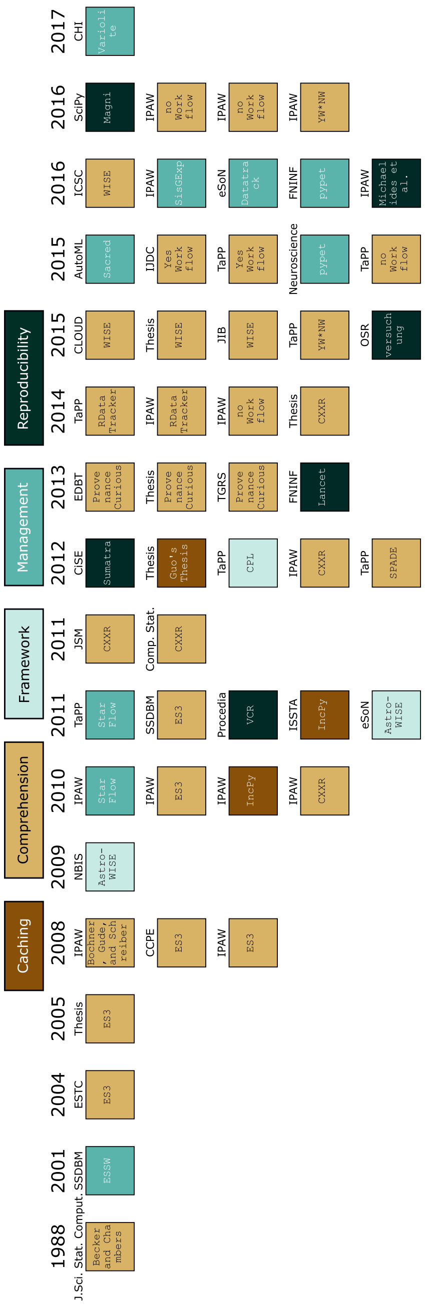

Fig. 9. Selected papers in Snowballing.

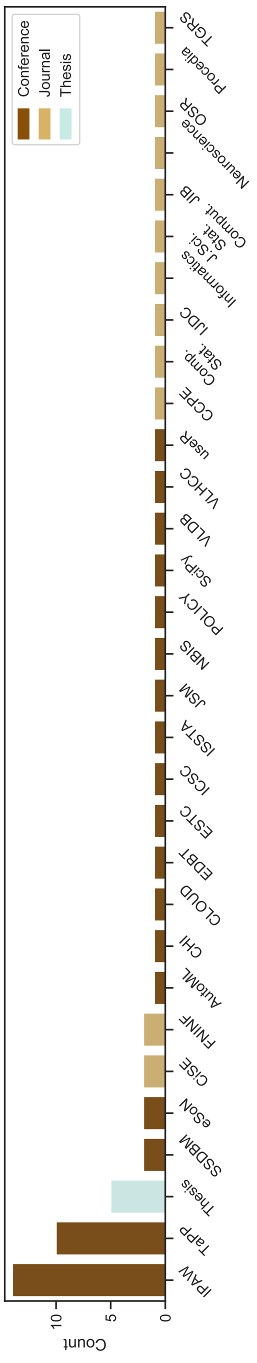

Fig. 10. Distribution of work by publishing location.

Table 1. Selected Approaches with Provenance Support: Main and Secondary Goals

Approach	Main goal	Secondary goals				
		Cache	Compr	Frame	Manag	Repro
Astro-WISE [99, 100]	Framework	✓	✓	✓	✗	✓
Becker and Chambers [11][3]	Comprehension	✗	✓	✗	✗	✓
Bochner et al. [12]	Comprehension	✗	✓	✓	✗	✗
CPL [82]	Framework	✗	✗	✓	✗	✗
CXXR [112–114, 116, 117][4]	Comprehension	✗	✓	✗	✗	✗
Datatrack [34]	Management	✗	✓	✗	✓	✗
ES3 [39, 41–44, 125][4]	Comprehension	✗	✓	✗	✗	✗
ESSW [40][4]	Management	✗	✓	✗	✓	✗
IncPy [53–55]	Caching	✓	✗	✗	✗	✗
Lancet [121]	Reproducibility	✗	✓	✗	✓	✓
Magni [103][4]	Reproducibility	✗	✗	✓	✗	✓
Michaelides et al. [90][4]	Reproducibility	✗	✓	✗	✗	✓
noWorkflow [98, 106, 107, 109]	Comprehension	✗	✓	✗	✓	✓
Provenance Curious [60–62][4]	Comprehension	✗	✓	✗	✗	✗
pypet [88, 89]	Management	✗	✗	✗	✓	✗
RDataTracker [74, 75]	Comprehension	✗	✓	✗	✗	✗
Sacred [51]	Management	✗	✓	✗	✓	✗
SisGExp [26]	Management	✗	✓	✗	✓	✓
SPADE [124]	Comprehension	✗	✓	✗	✗	✓
StarFlow [7, 8]	Management	✓	✓	✗	✓	✓
Sumatra [27][4]	Reproducibility	✗	✓	✗	✓	✓
Variolite [68]	Management	✗	✓	✗	✓	✓
VCR [45][4]	Reproducibility	✗	✓	✗	✗	✓
versuchung [30]	Reproducibility	✗	✓	✓	✗	✓
WISE [1–4]	Comprehension	✗	✓	✗	✗	✗
YesWorkflow [86, 87]	Comprehension	✗	✓	✗	✗	✗
YW*NW [29, 105]	Comprehension	✗	✓	✗	✓	✓
Main Goal / Total		1 / 3	11 / 23	2 / 5	7 / 11	6 / 14

Labels in secondary goals column refer to goals: *Cache*—Caching; *Compr*—Comprehension; *Frame*—Framework; *Manag*—Management; *Repro*—Reproducibility.

Table 1 presents the final selection of approaches with their papers. In this table, we categorized the approaches by their usage goals for provenance to answer **RQ1.1** (i.e., for what purpose do these tools collect provenance?). We identified five usage goals by reading the paper's motivations: caching, comprehension, framework, management, and reproducibility. For approaches that did not clearly specify the usage goals, we inferred by the proposed features.

The caching category represents approaches that use provenance for cache invalidation and that support reusing previous results. The comprehension category represents approaches that use provenance for understanding experiments, debugging scripts, documenting processes, checking compliance with standards, and auditing processes. The framework category represents approaches that propose generic mechanisms that allow others to implement their provenance systems. The management category represents approaches that use provenance for managing

[3]The authors indicated the software is of historical interest only and did not validate the classification.
[4]The authors did not reply.

experiments. Finally, the reproducibility category represents approaches that support reproducing, repeating, and comparing repetitions of experiments.

The most supported usage goals in the approaches are comprehension, reproducibility, and management, in this order. We also identified the main usage goal described in the papers. In this case, the order is comprehension, management, and reproducibility. Colors in Figure 9 represent the main usage goals of the approaches.

We grouped papers according to their publishing place to answer **RQ1.2** (i.e., Where and when were these tools published?). We identified 42 papers published in conferences, 14 articles in journals, and 5 theses. Figure 10 presents the distribution of work by publishing location. International Provenance and Annotation Workshop (IPAW) and Workshop on Theory and Practice of Provenance (TaPP) seem to be the preferred conferences. Computing in Science & Engineering (CiSE) and Frontiers in Neuroinformatics (FNINF) seem to be the preferred journals.

This section is structured in four subsections. Section 3.1 seeks to answer **RQ1.3** (i.e., how do these tools collect provenance?) by relating provenance applications to provenance types and classifying approaches according to our taxonomy. Section 3.2 seeks to answer **RQ1.4** (i.e., how do these tools manage provenance?) by relating provenance applications to storage, distribution, and versioning. Section 3.3 seeks to answer **RQ1.5** (i.e., how do these tools analyze provenance?) by relating provenance applications to visualization and query support. Finally, Section 3.4 discusses threats to the validity of the presented results.

RQ1.1. *For what purpose do these tools collect provenance?*
Answer: We identified five purposes for provenance: caching, comprehension, framework, management, and reproducibility. The most supported purposes are comprehension, reproducibility, and management.
Implications: Few approaches define frameworks for provenance and fewer approaches use provenance for caching. Moreover, we could not find any approach that collects provenance from scripts for security. All of these goals present opportunities for future research.

RQ1.2. *Where and when were these tools published?*
Answer: Most approaches were published in conferences, more specifically at IPAW and TaPP. The first approach that collects provenance from scripts was published in 1988, but the topic started to get more attention from 2008 on, due to the provenance challenges, and the number of approaches increased.
Implications: These results indicate which venues are interested in the topic and that the topic is attracting attention from the international community.

3.1 Provenance Collection

In this section, we categorize the approaches in diverse groups to answer **RQ1.3** (i.e., how do these tools collect provenance?). As we mentioned before, we identified 27 approaches that collect provenance from scripts. While the Earth Science System Server (ES3) [39, 41–44, 125] collects provenance from binary executions by default, it does include a plugin to collect provenance from IDL scripts optionally. Thus, it appears in our snowballing. Similarly, SPADE [46–49, 93, 124] has both reporters to collect operating system provenance and reporters to collected provenance from scripts compiled by an LLVM compiler. Hence, it also appears in this work.

Different approaches support different scripting languages. Table 2 relates supported languages to approaches. Some approaches appear multiple times in this table (i.e., support multiple scripting languages): the Core Provenance Library [82] (CPL) is a general-purpose provenance library with implementations for Python, R, C, CPP, and Java; and Gavish and Donoho [45] provide Verifiable Computational Result (VCR) implementations in R, Python, and Matlab. Besides these approaches,

Table 2. Supported Scripting Languages

Language	Count	Approaches
Python	16	Astro-WISE, Bochner et al. [12], CPL, IncPy, Lancet, Magni, noWorkflow, Provenance Curious, pypet, Sacred, StarFlow, Sumatra, VCR, versuchung, WISE, YW*NW
R	6	CPL, CXXR, Datatrack, RDataTracker, SisGExp, VCR
Agnostic	3	Sumatra, Variolite, YesWorkflow
S	1	Becker and Chambers [11]
IDL	1	ES3
Perl	1	ESSW
Blockly	1	Michaelides et al. [90]
LLVM	1	SPADE
Matlab	1	VCR

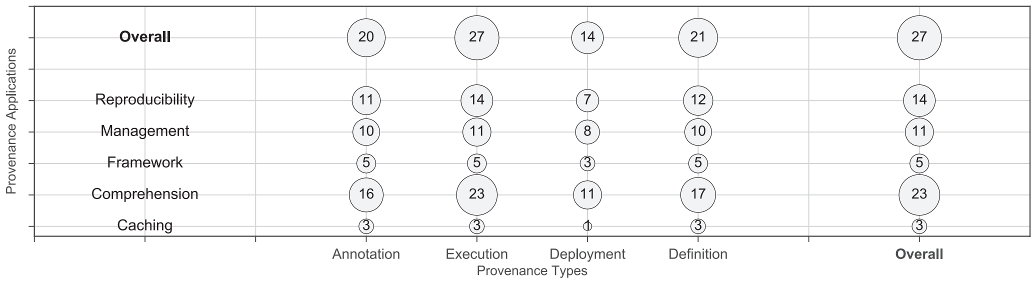

Fig. 11. Provenance types related to supported provenance applications. The numbers in the bubbles represent the number of approaches that support the feature. A single approach can appear in multiple bubbles.

three approaches (Sumatra [27], Variolite [68], and YesWorkflow [86, 87]) are language agnostic. That is, they support any scripting language that uses text files. As stated before, SPADE proposes a semi-agnostic approach that collects provenance from any scripting language that can be compiled by an LLVM compiler. Finally, in this table, we can see that the most common supported languages are Python and R. These languages are supported by 16 and 6 approaches, respectively.

As described in Section 2.1, we classify provenance into four types: annotations, execution, deployment, and definition. Figure 11 relates each provenance type to the supported provenance applications. We can observe interesting aspects of this figure. First, for comprehension, all approaches collect execution provenance. It indicates that these approaches focus on comprehending the execution. Second, most approaches seem to rely on annotations for collection. Finally, few approaches collect deployment provenance. Hence, the other approaches do not seem to consider the impact of the environment on their usage goals. In approaches that seek to support reproducibility, it might cause issues.

In Section 2.1, we described diverse mechanisms for collecting each type of provenance. We classify annotations according to their placement, extraction, inclusiveness, target, and necessity. We classify execution provenance collection into four groups: passive monitoring, overriding, post-mortem, and instrumentation. We classify deployment provenance according to the frequency of collection: single snapshot or continuous. Finally, we classify definition provenance collection according to how and when they collect it: collecting files in a directory (how/reading), extracting annotations or structure from scripts (how/parsing), collecting definition before or after trials (when/static), and collecting definition on demand (when/dynamic). Table 3 compares these provenance collection mechanisms for the approaches.

Even though Sumatra [27] is presented as language-agnostic in Table 2, it supports extracting Python modules and provides an API for extending to other languages. In addition to Python modules, Sumatra also collects a snapshot of environment variables from any scripting language as deployment provenance. Sumatra uses the post-mortem strategy for execution provenance collection. It collects files after the execution in a specific directory as outputs of a trial and the content of files and experiment before the execution as input. Additionally, Sumatra accepts external inclusive annotations on provenance to describe what is happening in the environment or experiment. Similar to Sumatra, SisGExp [26] and Variolite [68] support annotations on provenance to describe specific trials. However, while Sumatra focuses on guaranteeing the reproducibility of experiments, SisGExp and Variolite focus on managing multiple trials of experiments. Thus, these approaches do not collect deployment provenance. Both approaches use annotations not only on provenance but also to indicate what they should collect. SisGExp requires users to indicate the scripts and which files they want to collect. Thus, it applies the instrumentation strategy for execution provenance collection and collects definition provenance by reading the declared files. On the contrary, Variolite is a text editor plugin that uses external annotations referring to internal parts of scripts to collect variant versions as definition provenance. Variolite also employs the post-mortem strategy to collect execution provenance.

Many other approaches support inclusive annotations in scripts to assist provenance collection. Astro-WISE [99, 100], Bochner et al. [12], Datatrack [34], ESSW [40], Lancet [121], Magni [103], pypet [88, 89], Sacred [51], VCR [45], and versuchung [30] define libraries for provenance collection during execution. Thus, they use internal executable annotations to include provenance. Astro-WISE and versuchung propose embedded domain-specific languages for defining which objects should be traced in a descriptive way. Bochner et al. [12] and Magni propose generic functions for provenance collection and storage. Hence, programmers can use them as frameworks to implement other provenance tools. The same does not apply for other approaches, since they restrict their collection domain. Datatrack provides wrapper functions to collect dependencies among file accesses during the execution. ESSW provides Perl wrappers for the execution of external processes and functions and for defining file dependencies. Lancet uses annotations to describe experiments in a declarative way in Python. Sacred and pypet use annotations to declare parameters and outputs that should be collected. Sacred also uses the overriding strategy to collect the standard output. VCR uses annotations to log, load, and compute verifiable computational results with provenance.

In addition to the mandatory annotations that target the definition to assist provenance collection, Datatrack and pypet also support optional annotations that target the provenance, by passing extra parameters to the mandatory annotations functions. These extra parameters allow users to describe the collected provenance.

While all these approaches use annotations to collect execution provenance, the same cannot be said for other types of provenance. VCR supports only execution provenance. In addition to execution provenance, Bochner et al. [12], Datatrack and Magni provide functions for collecting deployment provenance continuously during the execution. However, Lancet and Sacred automatically collect a snapshot of the deployment provenance. Astro-WISE, Bochner et al. [12], ESSW, Sacred, and versuchung collect the definition of input files and output files as definition provenance dynamically, in addition to execution provenance. Astro-WISE, Magni, Sacred, and versuchung also collect the script source code as definition provenance. Similarly, Lancet collects the experiment declaration as definition provenance. The pypet approach can be integrated with Sumatra for definition and deployment provenance collection.

Table 3. Provenance Collection Strategies

Approach	Granularity	Annotations					Execution				Depl.		Definition			
		Placement	Extraction	Inclusiveness	Target	Necessity	Passive Monitoring	Overriding	Post-Mortem	Instrumentation	Snapshot	Continuous	Reading	Parsing	Static	Dynamic
Astro-WISE	User defined, Attributes, Files (I/O), Parameters, Source	Inte	Exec	Incl	Defi	Man	✗	✗	✗	✓	✗	✗	✓	✗	✗	✓
Becker and Chambers [11]	Commands, Variables, Random Seed	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓
Bochner et al. [12]	User defined, Files, Platform	Inte	Exec	Incl	Defi	Man	✗	✗	✗	✓	✗	✓	✓	✗	✗	✓
CPL	N/A	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
CXXR	Commands, Variables, Random Seed, Files (I)	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
Datatrack	User defined, Parameters, Platform, Modules	Inte	Exec	Incl	Defi Prov	Man Opt	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗
ES3	Files (I/O - metadata)	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
ESSW	User defined, Processes, Files (I/O)	Inte	Exec	Incl	Defi	Man	✗	✗	✗	✓	✗	✗	✓	✗	✗	✓
IncPy	Functions, Globals, Stack, Output, Files (I/O)	Inte	Exec	Incl Excl	Defi	Opt	✗	✓	✗	✓	✗	✗	✓	✓	✓	✓
Lancet	Arguments, Commands, Platform, Env. Var.	Inte	Exec	Incl	Defi	Man	✗	✗	✗	✓	✓	✗	✗	✓	✓	✗
Magni	User defined, Stack Trace, Platform, Source	Inte	Exec	Incl	Defi	Man	✗	✗	✗	✓	✗	✓	✓	✗	✓	✗
Michaelides et al. [90]	Blocks, Calls, Random Seed, User Input	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓
noWorkflow	Functions, Variables, Env. Var., Platform, Modules, Files (I/O)	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓	✗	✓	✓	✓	✓
Provenance Curious	Language Constructs, Files (I/O)	Exte	Pars	Incl	Defi	Man	✗	✗	✓	✗	✗	✗	✗	✓	✓	✗
pypet	Arguments, Output, Sumatra	Inte	Exec	Incl	Defi Prov	Man Opt	✗	✗	✗	✓	✓	✗	✓	✗	✓	✗
RDataTracker	Commands, Variables, Values, Env. Var., Platform, Modules, Files (I/O)	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓	✓	✓	✗	✗	✓
Sacred	User defined, Output, Modules, Host, Source, Files (I/O)	Inte	Exec	Incl	Defi	Opt	✗	✓	✗	✓	✓	✗	✓	✗	✓	✓
SisGExp	User defined, Files (I/O), Source	Exte	Pars	Incl	Prov	Man	✗	✗	✗	✓	✗	✗	✓	✗	✓	✗
SPADE	Functions, Returns, Arguments, Stack Trace, Env. Var.	Exte	Pars	Excl	Defi	Opt	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗
StarFlow	Functions, Modules, Files (I/O), Stack Trace	Inte	Pars Exec	Incl	Defi	Opt	✓	✓	✗	✓	✓	✗	✗	✓	✓	✓
Sumatra	Modules, Files (I/O)	Exte	Pars	Incl	Prov	Opt	✗	✗	✓	✗	✓	✗	✓	✗	✓	✗
Variolite	Arguments, Output, Source	Exte	Pars	Incl	Prov	Opt	✗	✗	✓	✗	✗	✗	✓	✗	✓	✗
VCR	User defined, Variables, Calls, Stack Trace	Inte	Exec	Incl	Defi	Man	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
versuchung	User defined, Files (I/O), Source	Inte	Exec	Incl	Defi	Man	✗	✗	✗	✓	✗	✗	✓	✗	✓	✓
WISE	Processes, Modules, Files (I/O - metadata)	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓	✗	✓	✗	✓	✗
YesWorkflow	User defined	Inte Exte	Pars	Incl	Defi	Man	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗
YW*NW	Variables, Dependencies, User defined	Inte	Pars	Incl	Defi	Man	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓

Labels in Annotations columns refer to categories described in Section 2.1.1 *Exte* —External; *Inte*—Internal; *Pars*—Parseable; *Exec*—Executable; *Incl*—Inclusive; *Excl*—Exclusive; *Defi*—Definition; *Prov*—Provenance; *Man*—Mandatory; *Opt*—Optional.

StarFlow [7, 8] also proposes a library that provides inclusive internal annotations for provenance collection. However, different from the other approaches, annotations are not the only mechanism for provenance collection in StarFlow. Instead, it uses specific naming conventions for arguments in function definitions and decorators in Python as annotations for provenance collection. These annotations are both parseable and executable as they are valid Python constructs. With these annotations, StarFlow statically decides in which order it should call the annotated functions based on declared file dependencies. Thus, StarFlow parses the scripts to extract annotations and collect definition provenance. During the execution, StarFlow also applies the passive monitoring strategy to extract function calls and the overriding strategy to collect file accesses. However, its execution provenance is independent of the annotations. StarFlow supports using execution provenance for verifying if annotations are correct. Hence, it uses optional annotations to extract the experiment pipeline and manage its execution. StarFlow supports distributing the pipeline to a cluster and re-executing only necessary functions if an input file changes. For keeping all dependencies during distribution, StarFlow also collects a snapshot of the modules as deployment provenance.

Similar to StarFlow, YesWorkflow [86, 87] also uses parseable internal annotations to extract pipelines from scripts. However, instead of using existing script constructs as annotations, YesWorkflow uses a domain specific language on commentaries. Thus, it is able to support almost all scripting languages but loses the ability to manage the execution and verifying if the annotations really represent the script definition. In addition to commentaries in a script, YesWorkflow annotations can also appear in external files referring to the script. YesWorkflow also uses annotations to determine URI templates to input and output files. After the execution, YesWorkflow applies the post-mortem strategy and collects all metadata from files that match these URI templates.

Provenance Curious [60–62] uses annotations to include data that is not collected during execution. It collects definition provenance by parsing Python scripts and collects execution provenance using the post-mortem strategy. Provenance Curious uses statistical models to infer all the provenance. It allows users to change the parsed definition provenance through external annotations to improve the inference.

The Workflow Instrumentation for Structure Extraction (WISE) [1–4] also uses the post-mortem strategy to collect accessed files. However, instead of considering only the definition provenance from the post-mortem strategy, it combines the post-mortem strategy with the overriding strategy. WISE overrides the script and its modules to collect provenance. It applies the post-mortem strategy for each invoked program to identify output files. WISE backups the original scripts and modules as definition provenance.

IncPy [53–55] and SPADE [46–49, 93, 124] use annotations to filter the provenance collection. Both approaches apply the overriding strategy to collect function calls and annotations to filter them. IncPy modifies the Python interpreter to collect provenance for caching. By default, it caches only pure functions (i.e., functions whose return values depend only on parameters and that do not cause side effects), files produced by these functions, and global variables. However, it allows users to decorate functions with internal annotations to force caching impure functions or to exclude pure functions from caching. As definition provenance, IncPy reads scripts definition and collects accessed file contents during execution for caching, and parses scripts to extract function dependencies for caching invalidation. Conversely, SPADE instruments scripts compiled with an LLVM compiler to provide comprehension. Thus, they only use external SPADE filters to exclude function calls. In addition to the execution provenance, SPADE supports the collection of a snapshot of environment variables as deployment provenance.

Becker and Chambers [11], CXXR [112–114, 116, 117], ES3 [43], Michaelides et al. [90], noWorkflow [98, 106–109], and RDataTracker [74–76] collect provenance without annotations, through the overriding strategy. Similar to WISE, ES3 also modifies IDL scripts and RDataTracker modifies R scripts to include instrumented functions for logging and overrides built-in functions. Becker and Chambers [11], CXXR, and Michaelides et al. [90] modify the interpreter for provenance collection. Since Michaelides et al. [90] have the goal of supporting reproducibility, they also collect definition provenance during execution according to what was executed. Thus, they unfold loops and replace user inputs with values. Similarly, Becker and Chambers [11] also collect the sequence of statements as definition provenance for supporting reproducibility. In addition to the overriding strategy, RDataTracker uses the passive monitoring strategy to collect inputs and outputs and information about top-level R statements. Similarly, noWorkflow combines the passive monitoring strategy with the overriding strategy for execution provenance collection. However, instead of changing the script or the interpreter, it defines custom profilers and tracers before the execution to track executed functions and lines and overrides only built-in functions to collect accessed files. Both noWorkflow and RDataTracker also collect the used scripts as definition provenance, and the imported modules and environment variables as deployment provenance.

YW*NW [29, 105] combines YesWorkflow and noWorkflow to use the annotations of the former as filters for variables and functions collected by the latter. This way, it uses parseable internal annotations together with automatic provenance collection mechanisms to collect all types of execution, deployment, and definition provenance.

Finally, CPL has no classification in Table 3. When we contacted the authors, they indicated that the classification is orthogonal to CPL. CPL was designed as a library to be used with other provenance tools instead of as a tool to collect provenance. Thus, the provenance collection strategies on CPL varies according to the tools that use it.

RQ1.3. *How do these tools collect provenance?*
Answer: The most commonly-used strategy for collecting execution provenance is to instrument the code with inclusive annotations. These annotations often appear inside the script definitions and are pervasive for all identified provenance usages. Some approaches also use annotations to collect deployment and definition provenance. However, the most used strategy for deployment provenance collection is taking a snapshot of automatically discovered dependencies and environment variables. Additionally, the most used strategy for definition provenance collection is statically reading files before the execution.
Implications: Few approaches employ a fully automated provenance collection that supports passive monitoring, overriding, and post-mortem strategies. This results in more work for users, which may hamper their adoption of provenance tools. Additionally, very few approaches support the dynamic collection of deployment provenance. Hence, most approaches are not suited for scripts that modify the environment during execution.

3.2 Provenance Management

In this section, we categorize the approaches related to how they store, distribute, and version provenance to answer **RQ1.4** (i.e., how do these tools manage provenance?). We already presented approaches that support reproducibility in Table 1. As we describe in Section 2.2, approaches store provenance in databases systems, memory, or files, and distribute the provenance through local files and remote repositories or web servers. Additionally, approaches may support versioning by identifying trials, storing sequences of trials, or storing the actual evolution intention. Table 4 compares the provenance management for the approaches.

Table 4. Provenance Management Classification

Approach	Artifacts	Storage			Dist.		Versioning
		Database	Memory	File	Local	Remote	
Astro-WISE	Oracle	✓	✗	✗	✗	✗	Sequence
Becker and Chambers [11]	Proprietary, Source	✗	✗	✓	✓	✗	✗
Bochner et al. [12]	PReServ	✗	✗	✗	✗	✓	✗
CPL	MySQL, PostgreSQL, 4store	✓	✗	✗	✓	✗	Trial ID
CXXR	Memory	✗	✓	✗	✗	✗	✗
Datatrack	VCS, Proprietary (CSV)	✗	✗	✓	✓	✓	Trial ID
ES3	XML Server, GraphML, Graphviz	✓	✗	✗	✓	✗	✗
ESSW	MySQL, Content DB, Graphviz	✓	✗	✓	✓	✗	Trial ID
IncPy	Content DB	✗	✗	✓	✓	✗	✗
Lancet	Log	✗	✗	✓	✓	✗	Intention
Magni	Proprietary (JSON, HDF5)	✗	✗	✓	✓	✗	Intention
Michaelides et al. [90]	Proprietary (INPWR), PROV, Source	✗	✗	✓	✓	✗	✗
noWorkflow	Content DB, SQLite, Prolog	✓	✗	✓	✓	✗	Intention
Provenance Curious	SQLite, GraphML	✓	✗	✗	✓	✗	✗
pypet	Proprietary (HDF5)	✗	✗	✓	✓	✗	Intention
RDataTracker	PROV-JSON	✗	✗	✓	✓	✗	Trial ID
Sacred	MongoDB, Relational, JSON	✓	✗	✓	✓	✗	Trial ID
SisGExp	PostgreSQL, Repository	✓	✗	✓	✗	✓	✗
SPADE	PostgreSQL, MySQL, H2, Neo4j, Datalog, GraphViz, PROV	✓	✗	✓	✓	✓	✗
StarFlow	OPM, Proprietary (CSV)	✗	✗	✓	✓	✗	✗
Sumatra	SQLite, VCS	✓	✗	✓	✗	✓	Intention
Variolite	Proprietary (JSON)	✗	✗	✓	✓	✗	Intention
VCR	Log, Repository	✗	✗	✓	✗	✓	✗
versuchung	Content DB, SQLite, Proprietary (Dict)	✓	✗	✓	✓	✗	Intention
WISE	Graphviz, GraphML	✗	✗	✓	✓	✗	✗
YesWorkflow	PROV, Datalog, Graphviz	✗	✗	✓	✓	✗	✗
YW*NW	noWorkflow + YesWorkflow	✓	✗	✓	✓	✗	✗

Figure 12 compares supported provenance applications to storage systems. The most popular storage systems are relational databases, proprietary files, and content databases. Note that we classify version control systems as content databases. Some approaches use more than one storage system. For instance, Sumatra uses both a version control system (content database) and a relational database. Thus, it appears as File and Database. Eleven approaches use relational databases: Astro-WISE uses Oracle; CPL and SPADE use MySQL or PostgreSQL; Provenance Curious, noWorkflow, Sumatra, versuchung, and YW*NW use SQLite; ESSW uses MySQL; SisGExp uses PostgreSQL. Sacred supports a variety of relational databases through an ORM system. Among approaches that use database systems, CPL and SPADE can also store provenance in graph databases (4store and Neo4j, respectively) instead of in relational databases. Finally, as NoSQL databases, ES3 uses XML Database Servers and Sacred supports MongoDB.

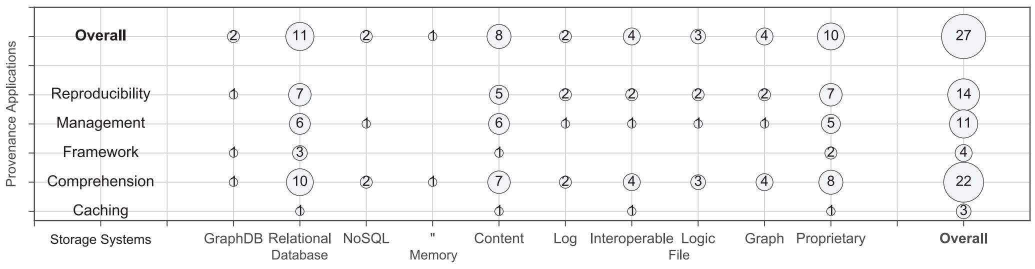

Fig. 12. Storage systems related to supported provenance applications. The numbers in the bubbles represent the number of approaches that support the feature. A single approach can appear in multiple bubbles.

In addition to the relational database, Sumatra stores provenance in version control systems and ESSW, noWorkflow, and YW*NW store provenance in a content database. IncPy and versuchung also store provenance in content databases. While IncPy uses content databases based on naming conventions for caching (i.e., without an additional database for metadata), versuchung stores Python dictionaries with provenance in proprietary files.

Similar to Sumatra, Datatrack also uses version control systems. However, instead of using it with a relational database, it uses a proprietary CSV file. VCR also stores provenance as a log in a content database. It proposes its own provenance repository for reproducible computational results. Lancet also stores provenance in a log file.

StarFlow, SPADE, RDataTracker, and YesWorkflow produce interoperable provenance formats as the result of provenance collection. The former creates OPM files and the others create PROV files. As an alternative to OPM, StarFlow also supports producing XML and CSV files with a proprietary data model. Similarly, SPADE supports storing provenance in proprietary text files, GraphViz files with combined graphs, or logic files for Datalog, as an alternative to databases and PROV files. YesWorkflow also provides an alternative to PROV. It supports producing graph definition files for GraphViz in three visualization formats (process-centric, data-centric, and combined) and logic files for Datalog. In addition to SPADE and YesWorkflow, the only approaches that store provenance in graph definition files are WISE and YW*NW. WISE produces GraphViz and GraphML files. YW*NW produces GraphViz files with combined graphs.

In addition to Datatrack, SPADE, StarFlow, and versuchung, six other approaches produce proprietary files. Michaelides et al. [90] store provenance in an intermediate notation for provenance and workflow reproducibility (INPWR). INPWR supports an easy mapping to PROV or to Blockly scripts for reproducibility. Becker and Chambers [11] store provenance in an intermediate format that stores a list of statements and objects affected by each statement. Similar to INPWR, Becker and Chambers [11] support converting provenance back to S scripts for reproducibility. All the other approaches that store provenance in proprietary files use common file formats: Magni uses JSON and HDF5 files; pypet uses HDF5; Sacred and Variolite use JSON.

Besides files and database systems, note in Figure 12 that one approach, CXXR, does not store provenance in the persistent memory. Its provenance exists only during the execution. Finally, Bochner et al. [12] do not store the provenance nor keeps it only in the memory. Instead, it distributes it to a remote web server that manages the storage. SPADE also supports transferring the provenance to a remote server instead of storing it.

All approaches that only store provenance in files support distributing provenance locally by distributing these files [1–4, 7, 8, 11, 29, 30, 34, 46–49, 54, 68, 75, 76, 86–90, 93, 98, 103, 105–109, 121, 124]. Additionally, repository approaches distribute provenance through the repositories themselves [27, 34, 45]. However, these are not the only ways to distribute provenance. Diverse approaches also convert the stored provenance into interoperable formats and other formats

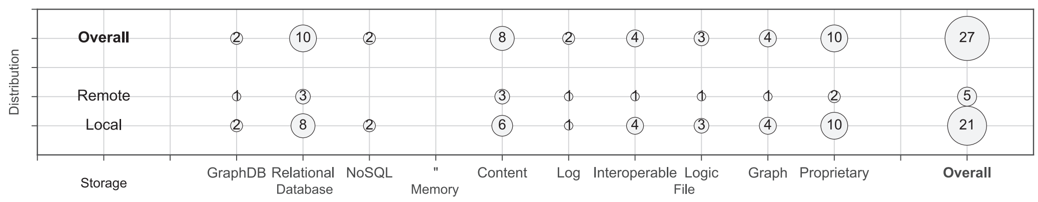

Fig. 13. Storage related to distribution. The numbers in the bubbles represent the number of approaches that support the feature. A single approach can appear in multiple bubbles.

suitable for analysis, as presented in Figure 13. This figure relates storage systems with distribution systems.

The approaches that store provenance in a database and produce files for distribution are CPL, ESSW, ES3, noWorkflow, Provenance Curious, Sacred, SPADE, versuchung, and YW*NW. CPL, Sacred, and versuchung produce files with a proprietary data model. ESSW produces GraphViz files for visualization. ES3 produces GraphViz and GraphML files. Provenance Curious produces GraphML files. noWorkflow produces GraphViz files for visualization and Prolog files for logic inference. Similarly, SPADE and YW*NW produce GraphViz files for visualization and Datalog files for logic inference. In addition to these files, noWorkflow's provenance can be distributed as a directory package [106]. Two approaches that store provenance in a database distribute it through repositories. One of them is the aforementioned Sumatra, that also stores provenance in repositories. The other is SisGExp, which provides a web server as a repository for experiments.

Fifteen out of 16 approaches that store provenance as files support distributing the same files [1–4, 7, 8, 11, 29, 30, 34, 46–49, 54, 68, 75, 76, 86–90, 93, 98, 103, 105–109, 124]. As stated before, ESSW does not distribute the same files, but it distributes graph files. In addition to distributing the proprietary INPWR format, Michaelides et al. [90] support transforming this file into PROV or back into executable Blockly scripts. Becker and Chambers [11] support transforming their proprietary files back into S scripts.

Astro-WISE, CPL, Datatrack, ESSW, Lancet, Magni, noWorkflow, pypet, RDataTracker, Sacred, Sumatra, Variolite, and versuchung support provenance versioning. As we stated before, Sumatra uses a version control system for provenance storage. Lancet provides optional functions to log the current definition version in version control systems. Hence, if a user commits each experiment, it is possible to keep track of provenance evolution. Similarly, Magni, pypet, and versuchung can be integrated with version control systems. Thus, in these systems, versions correspond to provenance versions. Since they use version control systems, they can record the evolution intention and they are able to compare files from different experiments.

Similar to the approaches that use version control systems, noWorkflow and Variolite also track the evolution intention by implementing their own versioning system for provenance. noWorkflow [106] assigns a trial version number for each trial, stores the evolution of trial versions, and supports restoring previous versions with intermediate provenance data. Variolite, however, supports creating variations of script definitions with branches in each variation.

Astro-WISE maintains a derivation reference for versioning. During the trial execution, it enforces the immutability of tracked objects and the uniqueness of object versions across all trials. It also allows an object of a newer trial to reference objects from previous trials. While this versioning strategy does not show the evolution intention, it allows users to have full provenance traceability.

Finally, CPL, DataTrack, ESSW, RDataTracker, and Sacred support a weak form of provenance evolution: These approaches just associate a trial identifier for each execution but do not track

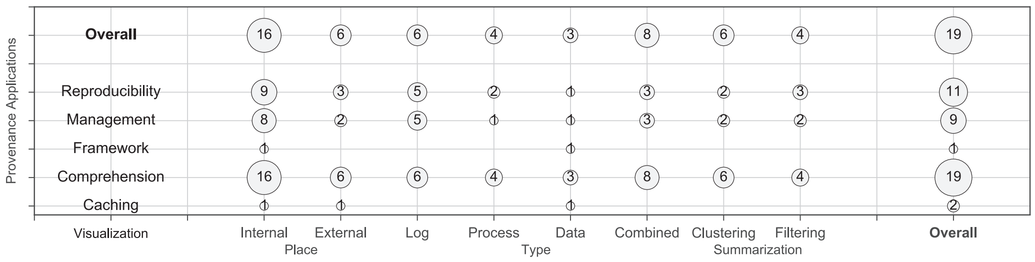

Fig. 14. Visualization related to supported provenance applications. The numbers in the bubbles represent the number of approaches that support the feature. A single approach can appear in multiple bubbles.

what motivated each trial evolution nor the actual evolution sequence, in case a user rolls back to a previous version. Trial sequences can be inferred in these approaches either by their moment of execution or by their identification sequence.

RQ1.4. *How do these tools manage provenance?*
Answer: Most approaches store provenance in relational databases, proprietary files, and content databases. The approaches that store provenance in files support sharing the provenance by sharing these files. Some approaches that store provenance in databases support converting the provenance to files for sharing. Finally, we identified approaches with versioning support. Some use full-fledged version control systems, others implement their own versioning, and some just provide basic versioning that identifies the id or timestamp of versions.
Implications: Most approaches that use files for sharing provenance do not support loading external provenance into the system. Moreover, using proprietary file formats makes the collaboration and concurrent work on projects harder. Using version control systems reduces these problems, but poses issues on how to structure the provenance.

3.3 Provenance Analysis

In this section, we categorize the approaches to answer **RQ1.5** (i.e., how do these tools analyze provenance?). We identify how each approach queries provenance (e.g., generic language or specific querying mechanism); how each approach visualizes provenance, according to the place of visualization, the type of visualization, and the support for summarization; and how each approach supports comparison (diff). Table 5 compares the approaches according to their provenance analysis.

Figure 14 relates provenance visualization with supported provenance applications. Note that only 19 approaches out of 27 support provenance visualization. As expected, users can employ all approaches that support provenance visualization for comprehension. Although only RDataTracker, SPADE, StarFlow [8], and YesWorkflow [87] store provenance in interoperable formats, Michaelides et al. [90] produce it for analysis and distribution. As we stated in Section 2.3, such files can be used to visualize and query provenance by external tools. Note that SPADE and YesWorkflow support both external and internal visualization mechanisms. These approaches provide their own visualization but also export provenance to interoperable files.

The most popular graph format for visualization are combined graphs. These graphs present both process and data and appear in eight approaches: ES3, ESSW, noWorkflow, Provenance Curious, RDataTracker, SPADE, YesWorkflow, and YW*NW. Among these approaches, five support summarizing the graph. noWorkflow and YW*NW use logic queries to select variables or function calls in the graph and filter everything that does not appear in the provenance of the selected elements. Provenance Curious [62] presents combined graphs that apply graph compression

Table 5. Provenance Analysis Classification, Based on Query, Visualization, and Diff

Approach	**Query**		**Visualization**								**Diff**	
			Place		**Type**				**Sum.**			
	Generic	**Specific**	**Internal**	**External**	**Log**	**Process**	**Data**	**Combined**	**Clustering**	**Filtering**	**Data**	**Provenance**
Astro-WISE	SQL	Functions, Web	✓	✗	✗	✗	✓	✗	✗	✗	✗	✓
Becker and Chambers [11]	✗	Functions	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗
Bochner et al. [12]	XQuery, XPath	Web	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
CPL	SPARQL, SQL	Functions	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
CXXR	✗	Functions	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Datatrack	✗	✗	✓	✗	✗	✗	✓	✗	✓	✗	✓	✗
ES3	XQuery	✗	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗
ESSW	SQL	Web	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗
IncPy	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓
Lancet	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
Magni	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
Michaelides et al. [90]	✗	PROV	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
noWorkflow	SQL, Prolog	Functions, Web	✓	✗	✓	✓	✗	✓	✓	✓	✗	✓
Provenance Curious	SQL	Functions	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗
pypet	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
RDataTracker	✗	DDG, PROV, Functions	✗	✓	✗	✗	✗	✓	✓	✓	✗	✓
Sacred	SQL	Web	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗
SisGExp	SQL	Web	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗
SPADE	SQL, Cypher, Datalog	PROV, Functions	✓	✓	✗	✗	✗	✓	✓	✓	✗	✓
StarFlow	✗	Functions, OPM	✗	✓	✗	✗	✗	✗	✗	✗	✓	✓
Sumatra	SQL	Command, Web	✓	✗	✓	✗	✗	✗	✗	✗	✓	✓
Variolite	✗	Command	✓	✗	✓	✗	✗	✗	✗	✗	✓	✗
VCR	✗	Web	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗
versuchung	SQL	Functions	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
WISE	✗	✗	✓	✗	✗	✓	✗	✗	✓	✗	✗	✗
YesWorkflow	Datalog	PROV	✓	✓	✗	✓	✓	✓	✗	✗	✗	✗
YW*NW	Datalog	✗	✓	✗	✗	✗	✗	✓	✗	✓	✗	✗

re-write rules for summarizations and support further manual summarizations. RDataTracker support clustering and filtering the graphs in the DDG explorer, an external tool designed to work with RDataTracker provenance. SPADE supports summarizing the graphs through provenance transformers.

In addition to the combined graphs, noWorkflow produces a process-centric graph that summarizes activations and support manual collapsing of nodes [98]. It also produces a trial evolution history graph and supports comparing the process-centric graphs of two trials [106]. Similarly, RDataTracker, SPADE, and Sumatra support comparing the provenance of trials. RDataTracker uses the DDG Explorer to compare lists of procedure nodes. SPADE compares responses to distributed provenance queries against cached prior responses. Sumatra compares provenance

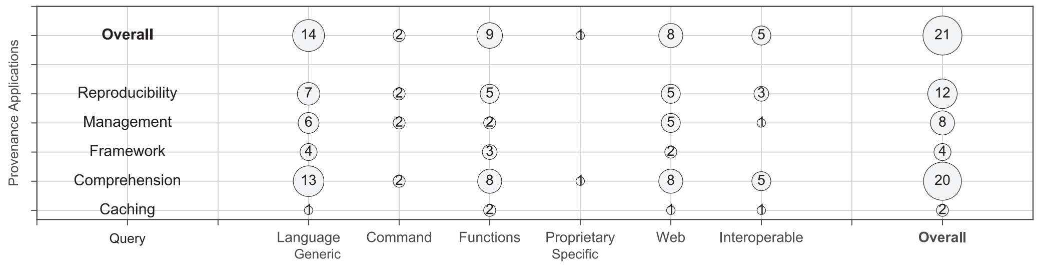

Fig. 15. Querying modes related to supported provenance applications. The numbers in the bubbles represent the number of approaches that support the feature. A single approach can appear in multiple bubbles.

through a web interface and command lines. It also allows comparing file definitions through the version control system. All approaches that support version control systems can use them to compare file definitions [30, 34, 88, 103, 121]. The same occurs with Variolite [68], that implements its own version control system. In addition to noWorkflow, RDataTracker, SPADE, and Sumatra, only Astro-WISE, Starflow, and IncPy compare provenance. Astro-Wise compares provenance to check which dependencies have changed. Starflow and Incpy, however, compares both provenance and data for cache invalidation. Both Sumatra and noWorkflow can also visualize trial provenance as a log. Similarly, Sacred, SisGExp, Variolite, and VCR also present the list of trials as a log.

YesWorkflow produces three types of graphs: combined graphs, process-centric, and data-centric. YW*NW [29, 105] uses YesWorkflow to filter noWorkflow variables. Thus, it produces graphs composed by YesWorkflow blocks, but with noWorkflow values. In addition to YesWorkflow and noWorkflow, the only approaches that support process-centric graphs are WISE, and the one proposed by Becker and Chambers [11]. WISE produces process-centric graphs with the sequence of external programs invoked by the scripts. It also supports producing summarized graphs that combine processes. Becker and Chambers [11] use commands to plot relationships between statements.

In addition to YesWorkflow, Datatrack and Astro-WISE produce data-centric views. While Datatrack produces a graph that combines all data accesses from the history into nodes, Astro-WISE produces a derivation tree and use parent relationships to describe provenance.

Figure 15 relates provenance querying with supported provenance applications. Note that 21 approaches out of 27 support provenance querying. All the aforementioned approaches that store or distribute provenance as interoperable formats support loading these files in standalone tools for specific provenance querying [7, 8, 46–49, 75, 76, 86, 87, 90, 93, 124].

The most popular querying mechanism is through query languages. In this category, we include general-purpose query languages and logic languages. The usage of query languages correlates to the high number of database systems for storage, as presented in Figure 12. Astro-WISE, ESSW, Provenance Curious, Sacred, SisGExp, Sumatra, noWorkflow, and versuchung support SQL queries, because they use relational databases. CPL supports SQL queries as well when it is using a relational database or SPARQL when it is using a graph database. Similarly, SPADE supports SQL when it is using a relational database, Cypher when it is using a graph database, or Datalog when it stores the provenance in Datalog files. ES3 [43] supports XQuery and XPath, since it uses an XML server. Bochner et al. [12] also support XQuery and XPath by sending queries to the remote server. In addition to these approaches, the approaches that support query languages are YesWorkflow and YW*NW. These approaches support Datalog queries. In addition to SQL queries, noWorkflow supports running Prolog queries.

The only approaches that use commands for querying are Sumatra and Variolite. Both of them use commands for querying annotations. Instead of commands, some approaches offer pre-defined functions for querying. Astro-WISE offers commands for obtaining files and provenance from previous executions. Becker and Chambers [11] use functions to query and visualize provenance stored in their proprietary files. Additionally, they use functions to recreate S scripts based on a set of executed statements. CPL provides functions to access the provenance and manage it in other provenance tools. CXXR uses functions to obtain lineage from memory. Note that CXXR does not store provenance at all. Thus, querying its provenance corresponds to obtaining command and variable lineage that is in the memory. noWorkflow supports using object models and methods to query the database in Jupyter Notebooks [109]. Provenance Curious uses functions to specify how should it try to infer provenance from output values. RDataTracker provides debugging functions that consume the provenance and supports querying functions in the DDG explorer. SPADE supports transformation functions for summarizations. StarFlow uses functions to determine which functions it should re-execute. StarFlow also uses functions for navigating in the lineage and discovering whether it should re-execute cached functions or not. The versuchung approach provides functions in the framework itself to analyze provenance as a new trial.

Finally, many approaches use web interfaces to facilitate provenance navigation, querying, and management [12, 26, 27, 40, 45, 51, 99, 100, 109]. While many specific provenance querying mechanisms have been proposed through functions, command, web interfaces, proprietary, and interoperable files, no specific query language has been proposed for scripts.

RQ1.5. *How do these tools analyze provenance?*
Answer: Most approaches support queries and visualizations for analyzing provenance. The most common querying modes are generic programming languages, specific functions, and web pages. No approach proposes a new querying language for provenance from scripts. The most common visualizations are implemented by the approaches themselves and display the provenance combining data and processes. Very few approaches support summarizing and comparing provenance.
Implications: Since most approaches define their own analysis methods, using a new approach involves learning a new tool. However, implementing a specific analysis tool for an approach allows performing more specific analyses.

3.4 Threats to Validity

Our systematic mapping has some threats to validity. Although we applied backward and forward snowballing exhaustively, the snowballing process does not guarantee that we discovered all related work. Additionally, our start set had papers published in only two distinct journals and three distinct conferences. This could lead to a disconnected component of a citation graph, which could concentrate only on a small niche. Note, however, that Jalali and Wohlin [65] suggest that there are no remarkable differences between database searches and backward snowballing, in the amount of obtained papers. Moreover, the number of papers in distinct conferences and journals we found indicate that our results did not concentrate in a small niche.

Since we considered only peer-reviewed work (e.g., paper, thesis), we left out unpublished related work. For instance, we did not include recipy (https://github.com/recipy/recipy), nor recordr (https://github.com/NCEAS/recordr), nor rdtLite (https://github.com/End-to-end-provenance/rdtLite) in our mapping, since they have no published papers. Recipy collects file provenance from Python scripts through a single import annotation that overrides built-in methods. Recordr collects file provenance from R scripts through a library annotation that overrides built-in functions. Additionally, recordr has functions to activate and deactivate the provenance collection for interactive sessions. Finally, rdtLite is a lightweight version of RDataTracker that uses the instrumentation strategy instead of the overriding strategy for execution provenance collection.

We considered only papers that we had access to their content and that matched our inclusion criteria. Out of 1,345 visited references, we could not access 9 papers, 20 papers were in different languages, 70 references were technical reports, 65 references were books, and 138 references were websites or email communications. Three papers that we could not access predates the first related approach [11], and they do not seem to be related to provenance according to their citation contexts and abstracts. We requested the other six to their authors, but we did not get a reply.

Another threat lies in the difficulty of identifying features and classifying papers. We excluded papers by reading just their abstracts and titles. Some papers could hide the support of provenance from scripts in the middle of the text. We believe we minimized the selection threat by keeping track and reading the place in which each citation appeared. However, we had some difficulty identifying whether some approaches were scripting provenance approaches, binary provenance approaches, or just had the benefits of provenance collection without the intention of collecting provenance.

To identify the features of the approaches, we have used information in the published papers and asked authors to validate our classifications based on a summarized version of the taxonomy. This leads to two extra threats. First, approaches for which we did not receive a reply may have other implemented features that were not described in the papers or may have evolved since the publication of the paper we surveyed here. Second, some authors that replied to our request had difficulties understanding the taxonomy based on the summarized version of the taxonomy. We attempted to reduce this threat by discussing the answers with the authors and sending them the Section 2 of this work.

4 CONCLUSIONS

In this work, we propose a taxonomy to characterize approaches that collect provenance from scripts, and we presented a systematic mapping with approaches that consider the structure of scripts to collect provenance. In this mapping, we identified five provenance applications, which these approaches support: caching, comprehension, framework, management, and reproducibility. Using these categories, we classified each approach according to their collection, management, and analysis support, according to the taxonomy we propose.

Regarding the taxonomy branches, we identified approaches that employ all mechanisms of provenance collection. However, few approaches collect fine-grained provenance in a transparent way (i.e., without demanding changes on the script). The only transparent approaches that collect fine-grained provenance are the one proposed by Becker and Chambers [11], CXXR [112, 117], noWorkflow [98, 106, 107, 109], RDataTracker [76], and the one proposed by Michaelides et al. [90]. Becker and Chambers [11] collect commands and variables in S. CXXR and RDataTracker collect commands and variables in R. noWorkflow collects variables and functions calls in Python. Michaelides et al. [90] collect block variables and block calls in Blockly. All these approaches have limited support for collecting the provenance of complex data structures. Hence, research is needed to develop efficient fine-grained provenance collection that supports complex data structures.

Few approaches use repositories to share execution and deployment provenance, while a considerable number of approaches use version control systems that allow sharing and comparing definition provenance. Moreover, many approaches support sharing provenance only by sharing the generated provenance files, without providing the means to compare or reuse such files. Thus, future research opportunities include proposing provenance distribution mechanisms for scripts that bundle everything that is necessary for reproducibility in packages [21] and that allows users to distribute provenance in repositories and compare not only definition provenance but also execution and deployment provenance.

While the approaches that use version control systems can track the intention of the experiment evolution, these systems are not adapted to track the intention according to the life cycle of experiments [85]. Hence, future research opportunities include proposing version control systems that differentiate the composition phase (i.e., the phase where scientists formulate hypothesis and compose execution plans) from the analysis phase (i.e., the phase where scientists query and visualize results, seeking to elaborate conclusions to confirm or refute the hypotheses of the experiment) of experiments, and that support the exploratory nature of experiments. The closest approaches that try to overcome these issues are noWorkflow and Sumatra. noWorkflow [106] proposes a version model for provenance collected from scripts. Sumatra [27] provides a layer on top of version control systems that adapts such tools for scientists.

Regarding provenance analysis, we found 14 approaches that support generic query languages and 19 approaches that support specific query mechanisms. As we mentioned before, none of these specific mechanisms is a query language defined for script provenance. Note, however, that 10 approaches provide custom functions for querying the provenance. While such functions are not query languages, they can also be considered specific query systems, and they could be embedded in a domain specific language. Moreover, we foresee the opportunity of developing provenance queries by example, using script slices. As for visualizations, we could only identify six approaches that support provenance clustering [2, 34, 61, 75, 98, 124] and only four that support graph filtering [75, 98, 105, 124]. It indicates an opportunity for future research to propose different summarization techniques, such as sampling. Moreover, the current provenance graphs are limited to directed graphs representing the provenance as-is. However, in the context of scripts, we foresee using provenance to represent different types of graphs, such as heat maps, indicating which parts of the scripts contribute more to a result, Sankey Diagrams, presenting dataflows with different flow sizes, and others.

REFERENCES

[1] Ruben Acuña. 2015. *Understanding Legacy Workflows through Runtime Trace Analysis.* Master's thesis. Arizona State University.

[2] Ruben Acuña, Jacques Chomilier, and Zoé Lacroix. 2015. Managing and documenting legacy scientific workflows. *J. Integr. Bioinformat.* 12, 3 (2015), 277–277.

[3] Ruben Acuña and Zoé Lacroix. 2016. Extracting semantics from legacy scientific workflows. In *Proceedings of the ICSC.* IEEE, 9–16.

[4] Ruben Acuña, Zoé Lacroix, and Rida A. Bazzi. 2015. Instrumentation and trace analysis for ad-hoc Python workflows in cloud environments. In *Proceedings of the CLOUD.* IEEE, 114–121.

[5] Ben Adida, Mark Birbeck, Shane McCarron, and Steven Pemberton. 2008. RDFa in XHTML: Syntax and processing. *W3C Prop. Recommend.* 7 (2008), 1–89.

[6] Manish Kumar Anand, Shawn Bowers, and Bertram Ludäscher. 2010. Provenance browser: Displaying and querying scientific workflow provenance graphs. In *Proceedings of the ICDE.* IEEE, 1201–1204.

[7] Elaine Angelino, Uri Braun, David A. Holland, and Daniel W. Margo. 2011. Provenance integration requires reconciliation. In *Proceedings of the TaPP.* USENIX, 1–6.

[8] Elaine Angelino, Daniel Yamins, and Margo Seltzer. 2010. StarFlow: A script-centric data analysis environment. In *Proceedings of the IPAW.* Springer, 236–250.

[9] Keith A. Baggerly and Kevin R. Coombes. 2009. Deriving chemosensitivity from cell lines: Forensic bioinformatics and reproducible research in high-throughput biology. *Ann. Appl. Stat.* 3, 4 (2009), 1309–1334.

[10] Zhuowei Bao, Sarah Cohen-Boulakia, Susan B. Davidson, and Pierrick Girard. 2009. PDiffView: Viewing the difference in provenance of workflow results. In *Proceedings of the VLDB.* VLDB Endowment, 1638–1641.

[11] Richard A. Becker and John M. Chambers. 1988. Auditing of data analyses. *SIAM J. Sci. Statist. Comput.* 9, 4 (1988), 747–760.

[12] Carsten Bochner, Roland Gude, and Andreas Schreiber. 2008. A Python library for provenance recording and querying. In *Proceedings of the IPAW.* Springer, 229–240.

[13] Uri Braun, Simson Garfinkel, David A. Holland, Kiran-Kumar Muniswamy-Reddy, and Margo I. Seltzer. 2006. Issues in automatic provenance collection. In *Proceedings of the IPAW*. Springer, 171–183.

[14] Steven P. Callahan, Juliana Freire, Emanuele Santos, Carlos Eduardo Scheidegger, Claudio T. Silva, and Huy T. Vo. 2006. Managing the evolution of dataflows with VisTrails. In *Proceedings of the ICDE*. IEEE, 71–71.

[15] Adriane Chapman and H. V. Jagadish. 2010. Understanding provenance black boxes. *Distrib. Parallel Databases* 27, 2 (2010), 139–167.

[16] Amit Chavan, Silu Huang, Amol Deshpande, Aaron Elmore, Samuel Madden, and Aditya Parameswaran. 2015. Towards a unified query language for provenance and versioning. In *Proceedings of the TaPP*. USENIX, 1–6.

[17] Artem Chebotko, John Abraham, Pearl Brazier, Anthony Piazza, Andrey Kashlev, and Shiyong Lu. 2013. Storing, indexing and querying large provenance data sets as RDF graphs in apache HBase. In *Proceedings of the IEEE SERVICES*. IEEE, 1–8.

[18] Artem Chebotko, Shiyong Lu, Xubo Fei, and Farshad Fotouhi. 2010. RDFProv: A relational RDF store for querying and managing scientific workflow provenance. *Data Knowl. Engineer.* 69, 8 (2010), 836–865.

[19] James Cheney, Amal Ahmed, and Umut A. Acar. 2011. Provenance as dependency analysis. *Math. Struct. Comput. Sci.* 21, 06 (2011), 1301–1337.

[20] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2007. Provenance in databases: Why, how, and where. *Found. Trends Databases* 1, 4 (2007), 379–474.

[21] Fernando Chirigati, Dennis Shasha, and Juliana Freire. 2013. Reprozip: Using provenance to support computational reproducibility. In *Proceedings of the TaPP*. USENIX, 977– 980.

[22] Pavan Kumar Chittimalli and Ravindra Naik. 2014. Variable provenance in software systems. In *Proceedings of the RSSE*. ACM, 9–13.

[23] Jon Claerbout and Martin Karrenbach. 1992. Electronic documents give reproducible research a new meaning. In *Proceedings of the SEG*. SEG, 601–604.

[24] Ben Clifford, Ian Foster, Jens-S. Voeckler, Michael Wilde, and Yong Zhao. 2008. Tracking provenance in a virtual data grid. *Concurr. Comput.: Pract. Exper.* 20, 5 (2008), 565–575.

[25] Flavio Costa, Vítor Silva, Daniel De Oliveira, Kary Ocaña, Eduardo Ogasawara, Jonas Dias, and Marta Mattoso. 2013. Capturing and querying workflow runtime provenance with PROV: A practical approach. In *Proceedings of the EDBT/ICDT*. ACM, 282–289.

[26] Sergio Manuel Serra da Cruz and José Antonio Pires do Nascimento. 2016. SisGExp: Rethinking long-tail agronomic experiments. In *Proceedings of the IPAW*. Springer, 214–217.

[27] Andrew Davison. 2012. Automated capture of experiment context for easier reproducibility in computational research. *Comput. Sci. Engineer.* 14, 4 (2012), 48–56.

[28] Brian Demsky. 2009. Garm: Cross application data provenance and policy enforcement. In *Proceedings of the HotSec*, vol. 9. USENIX, 10–10.

[29] Saumen Dey, Khalid Belhajjame, David Koop, Meghan Raul, and Bertram Ludäscher. 2015. Linking prospective and retrospective provenance in scripts. In *Proceedings of the TaPP*. USENIX, 1–7.

[30] Christian Dietrich and Daniel Lohmann. 2015. The dataref versuchung: Saving time through better internal repeatability. *SIGOPS Operat. Syst. Rev.* 49, 1 (2015), 51–60.

[31] David L. Donoho, Arian Maleki, Inam Ur Rahman, Morteza Shahram, and Victoria Stodden. 2009. Reproducible research in computational harmonic analysis. *Comput. Sci. Engineer.* 11, 1 (2009), 8–18.

[32] Chris Drummond. 2009. Replicability is not reproducibility: Nor is it good science. In *Proceedings of the ICML*. International Machine Learning Society, 1–4.

[33] Paul F Dubois. 1999. Ten good practices in scientific programming. *Comput. Sci. Engineer.* 1, 1 (1999), 7–11.

[34] Philip Eichinski and Paul Roe. 2016. Datatrack: An R package for managing data in a multi-stage experimental workflow. In *Proceedings of the eSoN*. IEEE, 1–8.

[35] Jacky Estublier. 2000. Software configuration management: A roadmap. In *Proceedings of the ICSE*. ACM, 279–289.

[36] Rosa Filguiera, Iraklis Klampanos, Amrey Krause, Mario David, Alexander Moreno, and Malcolm Atkinson. 2014. Dispel4Py: A Python framework for data-intensive scientific computing. In *Proceedings of the DISCS*. 9–16.

[37] Juliana Freire, David Koop, Emanuele Santos, and Cláudio T Silva. 2008. Provenance for computational tasks: A survey. *Comput. Sci. Engineer.* 10, 3 (2008), 11–21.

[38] Juliana Freire, Cláudio T. Silva, Steven P. Callahan, Emanuele Santos, Carlos E. Scheidegger, and Huy T. Vo. 2006. Managing rapidly-evolving scientific workflows. In *Proceedings of the IPAW*. Springer, 10–18.

[39] James Frew. 2004. Earth system science server (ES3): Local infrastructure for earth science product management. In *Proceedings of the ESTC*. NASA, 1–5.

[40] James Frew and Rajendra Bose. 2001. Earth system science workbench: A data management infrastructure for earth science products. In *Proceedings of the SSDBM*. IEEE, 180–189.

[41] James Frew, Greg Janée, and Peter Slaughter. 2010. Automatic provenance collection and publishing in a science data production environment—Early results. In *Proceedings of the IPAW*. Springer, 27–33.

[42] James Frew, Greg Janée, and Peter Slaughter. 2011. Provenance-enabled automatic data publishing. In *Proceedings of the SSDBM*. Springer, 244–252.

[43] James Frew, Dominic Metzger, and Peter Slaughter. 2008. Automatic capture and reconstruction of computational provenance. *Concurr. Comput.: Pract. Exper.* 20, 5 (2008), 485–496.

[44] James Frew and Peter Slaughter. 2008. Es3: A demonstration of transparent provenance for scientific computation. In *Proceedings of the IPAW*. Springer, 200–207.

[45] Matan Gavish and David Donoho. 2011. A universal identifier for computational results. *Procedia Comput. Sci.* 4 (2011), 637–647.

[46] Ashish Gehani, Hasanat Kazmi, and Hassaan Irshad. 2016. Scaling spade to "big provenance." In *Proceedings of the TaPP*. USENIX Association, 26–33.

[47] Ashish Gehani and Dawood Tariq. 2012. SPADE: Support for provenance auditing in distributed environments. In *Proceedings of the useR!*. Springer-Verlag, 101–120.

[48] Ashish Gehani and Dawood Tariq. 2014. Provenance-only integration. In *Proceedings of the TaPP*. USENIX, 1–8.

[49] Ashish Gehani, Dawood Tariq, Basim Baig, and Tanu Malik. 2011. Policy-based integration of provenance metadata. In *Proceedings of the POLICY*. IEEE, 149–152.

[50] Boris Glavic and Klaus R. Dittrich. 2007. Data provenance: A categorization of existing approaches. In *Proceedings of the BTW*. GI, 227–241.

[51] Klaus Greff and Jürgen Schmidhuber. 2015. Introducing sacred: A tool to facilitate reproducible research. In *Proceedings of the AutoML*. International Machine Learning Society, 1–6.

[52] Paul Groth, Simon Miles, and Luc Moreau. 2005. PReServ: Provenance recording for services. In *Proceedings of the AHM*, vol. 2005. EPSRC, 1–8.

[53] Philip Jia Guo. 2012. *Software Tools to Facilitate Research Programming*. Ph.D. Dissertation. Stanford University, Stanford University.

[54] Philip J. Guo and Dawson Engler. 2011. Using automatic persistent memoization to facilitate data analysis scripting. In *Proceedings of the ISSTA*. ACM, 287–297.

[55] Philip J. Guo and Dawson R. Engler. 2010. Towards practical incremental recomputation for scientists: An implementation for the Python language. In *Proceedings of the IPAW*. Springer, 1–10.

[56] Philip J. Guo and Dawson R. Engler. 2011. CDE: Using system call interposition to automatically create portable software packages. In *Proceedings of the ATC*. USENIX Association, 1–6.

[57] Philip J. Guo and Margo Seltzer. 2012. BURRITO: Wrapping your lab notebook in computational infrastructure. In *Proceedings of the TaPP*, vol. 12. USENIX, 1–7.

[58] Brooks Hanson, Andrew Sugden, and Bruce Alberts. 2011. Making data maximally available. *Science* 331, 6018 (2011), 649–649.

[59] Rinke Hoekstra and Paul Groth. 2014. PROV-O-Viz-understanding the role of activities in provenance. In *Proceedings of the IPAW*. Springer, 215–220.

[60] Mohammad Rezwanul Huq. 2013. *An Inference-based Framework for Managing Data Provenance*. Ph.D. Dissertation. University of Twente.

[61] Mohammad Rezwanul Huq, Peter M. G. Apers, and Andreas Wombacher. 2013. An inference-based framework to manage data provenance in Geoscience applications. *IEEE Trans. Geosci. Remote Sens.* 51, 11 (2013), 5113–5130.

[62] Mohammad Rezwanul Huq, Peter M. G. Apers, and Andreas Wombacher. 2013. ProvenanceCurious: A tool to infer data provenance from scripts. In *Proceedings of the EDBT*. ACM, 765–768.

[63] John P. A. Ioannidis. 2005. Why most published research findings are false. *PLOS Med.* 2, 8 (2005), e124.

[64] Keith R. Jackson. 2002. pyGlobus: A Python interface to the Globus toolkit. *Concurr. Comput.: Pract. Exper.* 14, 13–15 (2002), 1075–1083.

[65] Samireh Jalali and Claes Wohlin. 2012. Systematic literature studies: Database searches vs. backward snowballing. In *Proceedings of the ESEM*. ACM, 29–38.

[66] Matthew B. Jones, Bertram Ludäscher, Timothy McPhillips, Paolo Missier, Christopher Schwalm, Peter Slaughter, Dave Vieglais, Lauren Walker, and Yaxing Wei. 2016. DataONE: A data federation with provenance support. In *Proceedings of the IPAW*, vol. 9672. Springer, 230.

[67] Mary Beth Kery. 2017. Tools to support exploratory programming with data. In *Proceedings of the VL/HCC*. IEEE, 321–322.

[68] Mary Beth Kery, Amber Horvath, and Brad Myers. 2017. Variolite: Supporting exploratory programming by data scientists. In *Proceedings of the CHI*. ACM, 1–12.

[69] Donald E. Knuth. 1984. Literate programming. *Computer* 1, 2 (1984), 97–111.

[70] Troy Kohwalter, Thiago Oliveira, Juliana Freire, Esteban Clua, and Leonardo Murta. 2016. Prov viewer: A graph-based visualization tool for interactive exploration of provenance data. In *Proceedings of the IPAW*. Springer, 71–82.

[71] David Koop, Emanuele Santos, Bela Bauer, Matthias Troyer, Juliana Freire, and Cláudio T. Silva. 2010. Bridging workflow and data provenance using strong links. In *Proceedings of the SSDBM*, vol. 28. Springer, 397–415.

[72] Johannes Köster and Sven Rahmann. 2012. Snakemake—A scalable bioinformatics workflow engine. *Bioinformatics* 28, 19 (2012), 2520–2522.

[73] Hans Petter Langtangen. 2006. *Python Scripting for Computational Science* (3rd ed.), vol. 3. Springer, Berlin.

[74] Barbara Lerner and Emery Boose. 2014. POSTER: RDataTracker and DDG explorer. In *Proceedings of the IPAW*. Springer, 1–3.

[75] Barbara Lerner and Emery Boose. 2014. RDataTracker: Collecting provenance in an interactive scripting environment. In *Proceedings of the TaPP*. USENIX, 1–4.

[76] Barbara Lerner, Emery Boose, and Luis Perez. 2018. Using introspection to collect provenance in R. *Informatics* 5, 1 (2018), 12.

[77] Chunhyeok Lim, Shiyong Lu, Artem Chebotko, and Farshad Fotouhi. 2010. Prospective and retrospective provenance collection in scientific workflow environments. In *Proceedings of the SCC*. IEEE, 449–456.

[78] Chunhyeok Lim, Shiyong Lu, Artem Chebotko, Farshad Fotouhi, and Andrey Kashlev. 2013. OPQL: Querying scientific workflow provenance at the graph level. *Data Knowl. Engineer.* 88, 0 (2013), 37–59.

[79] Cui Lin, Shiyong Lu, Xubo Fei, Artem Chebotko, Darshan Pai, Zhaoqiang Lai, Farshad Fotouhi, and Jing Hua. 2009. A reference architecture for scientific workflow management systems and the VIEW SOA solution. *IEEE Trans. Services Comput.* 2, 1 (2009), 79–92.

[80] Ji Liu, Esther Pacitti, Patrick Valduriez, and Marta Mattoso. 2015. A survey of data-intensive scientific workflow management. *J. Grid Comput.* 13, 4 (2015), 457– –493.

[81] Clifford Lynch. 2000. Authenticity and integrity in the digital environment: An exploratory analysis of the central role of trust. *Council Library Info. Res.* 32, 1 (2000), 1–84.

[82] Peter Macko and Margo Seltzer. 2012. A general-purpose provenance library. In *Proceedings of the TaPP*. USENIX, 1–6.

[83] Anderson Marinho, Marta Mattoso, Claudia Werner, Vanessa Braganholo, and Leonardo Murta. 2011. Challenges in managing implicit and abstract provenance data: Experiences with ProvManager. In *TaPP*. USENIX, Heraklion, Crete, Greece, 1–6.

[84] Marta Mattoso, Jonas Dias, Kary A. C. S. Ocaña, Eduardo Ogasawara, Flavio Costa, Felipe Horta, Vítor Silva, and Daniel de Oliveira. 2015. Dynamic steering of HPC scientific workflows: A survey. *Future Gen. Comput. Syst.* 46 (2015), 100–113.

[85] Marta Mattoso, Claudia Werner, Guilherme Horta Travassos, Vanessa Braganholo, Eduardo Ogasawara, Daniel Oliveira, Sergio Cruz, Wallace Martinho, and Leonardo Murta. 2010. Towards supporting the life cycle of large scale scientific experiments. *Int. J. Bus. Process Integr. Manage.* 5, 1 (2010), 79–92.

[86] Timothy McPhillips, Shawn Bowers, Khalid Belhajjame, and Bertram Ludäscher. 2015. Retrospective provenance without a runtime provenance recorder. In *Proceedings of the TaPP*. USENIX, 1–7.

[87] Timothy McPhillips, Tianhong Song, Tyler Kolisnik, Steve Aulenbach, Khalid Belhajjame, Kyle Bocinsky, Yang Cao, Fernando Chirigati, Saumen Dey, Juliana Freire et al. 2015. YesWorkflow: A user-oriented, language-independent tool for recovering workflow information from scripts. *Int. J. Dig. Curat.* 10, 1 (2015), 298–313.

[88] Robert Meyer and Klaus Obermayer. 2015. pypet: A Python toolkit for simulations and numerical experiments. *Neuroscience* 16, Suppl 1 (2015), P184.

[89] Robert Meyer and Klaus Obermayer. 2016. pypet: A Python toolkit for data management of parameter explorations. *Front. Neuroinformat.* 10 (2016), 1–16.

[90] Danius T. Michaelides, Richard Parker, Chris Charlton, William J. Browne, and Luc Moreau. 2016. Intermediate notation for provenance and workflow reproducibility. In *Proceedings of the IPAW*. Springer, 83–94.

[91] Simon Miles, Paul Groth, Steve Munroe, and Luc Moreau. 2011. PrIMe: A methodology for developing provenance-aware applications. *ACM Trans. Software Engineer. Methodol.* 20, 3 (2011), 8.

[92] Paolo Missier, Saumen Dey, Khalid Belhajjame, Víctor Cuevas-Vicenttín, and Bertram Ludäscher. 2013. D-PROV: Extending the PROV provenance model with workflow structure. In *Proceedings of the TaPP*. USENIX, 1–7.

[93] Scott Moore, Ashish Gehani, and Natarajan Shankar. 2013. Declaratively processing provenance metadata. In *Proceedings of the TaPP*. USENIX, 1–8.

[94] Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers et al. 2011. The open provenance model core specification (v1. 1). *Future Gen. Comput. Syst.* 27, 6 (2011), 743–756.

[95] Luc Moreau, Bertram Ludäscher, Ilkay Altintas, Roger S. Barga, Shawn Bowers, Steven Callahan, George Chin, Ben Clifford, Shirley Cohen, Sarah Cohen-Boulakia et al. 2008. Special issue: The first provenance challenge. *Concurr. Comput.: Pract. Exper.* 20, 5 (2008), 409–418.

[96] Luc Moreau and Paolo Missier. 2012. PROV-DM: The PROV Data Model. W3C Proposed Recommendation. Retrieved from http://www.w3.org/TR/prov-dm.

[97] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo I. Seltzer. 2006. Provenance-Aware storage systems. In *Proceedings of the ATC*. USENIX Association, 43–56.

[98] Leonardo Murta, Vanessa Braganholo, Fernando Chirigati, David Koop, and Juliana Freire. 2014. noWorkflow: Capturing and analyzing provenance of scripts. In *Proceedings of the IPAW*. Springer, 71–83.

[99] Johnson Mwebaze, Danny Boxhoorn, and Edwin Valentijn. 2009. Astro-wise: Tracing and using lineage for scientific data processing. In *Proceedings of the NBIS*. IEEE, 475–480.

[100] Johnson Mwebaze, Danny Boxhoorn, and Edwin Valentijn. 2011. Dynamic pipeline changes in scientific data processing. In *Proceedings of the eSoN*. IEEE, 263–270.

[101] Wellington Oliveira, Daniel De Oliveira, and Vanessa Braganholo. 2018. Provenance analytics for workflow-based computational experiments: A survey. *Comput. Surveys* 51, 3 (2018), 53.

[102] John K. Ousterhout. 1998. Scripting: Higher level programming for the 21st century. *Computer* 31, 3 (1998), 23–30.

[103] Christian Schou Oxvig, Thomas Arildsen, and Torben Larsen. 2016. Storing reproducible results from computational experiments using scientific Python packages. In *Proceedings of the SciPy*. SciPy, 45–50.

[104] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. 2008. Systematic mapping studies in software engineering. In *Proceedings of the EASE*, vol. 8. ACM, 68–77.

[105] João Felipe Pimentel, Saumen Dey, Timothy McPhillips, Khalid Belhajjame, David Koop, Leonardo Murta, Vanessa Braganholo, and Bertram Ludäscher. 2016. Yin & Yang: Demonstrating complementary provenance from noWorkflow & YesWorkflow. In *Proceedings of the IPAW*. Springer, 161–165.

[106] João Felipe Pimentel, Juliana Freire, Vanessa Braganholo, and Leonardo Murta. 2016. Tracking and analyzing the evolution of provenance from scripts. In *Proceedings of the IPAW*. Springer, 16–28.

[107] João Felipe Pimentel, Juliana Freire, Leonardo Murta, and Vanessa Braganholo. 2016. Fine-grained provenance collection over scripts through program slicing. In *Proceedings of the IPAW*. Springer, 199–203.

[108] Joao Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2017. noWorkflow: A tool for collecting, analyzing, and managing provenance from Python scripts. *Very Large Data Bases* 10, 12 (2017), 1841–1844.

[109] João Felipe Nicolaci Pimentel, Vanessa Braganholo, Leonardo Murta, and Juliana Freire. 2015. Collecting and analyzing provenance on interactive notebooks: When IPython meets noWorkflow. In *Proceedings of the TaPP*. USENIX, 1–6.

[110] João Felipe N. Pimentel, Paolo Missier, Leonardo Murta, and Vanessa Braganholo. 2018. Versioned-PROV: A PROV extension to support mutable data entities. In *Proceedings of the IPAW*. Springer, 87–100.

[111] Raghu Ramakrishnan and Jeffrey D. Ullman. 1995. A survey of deductive database systems. *J. Logic Program.* 23, 2 (1995), 125–149.

[112] Andrew Runnalls and Chris Silles. 2012. Provenance tracking in R. In *Proceedings of the IPAW*. Springer, 237–239.

[113] Andrew R. Runnalls. 2011. Aspects of CXXR internals. *Comput. Stat.* 26, 3 (2011), 427–442.

[114] Andrew R. Runnalls and Chris A. Silles. 2011. CXXR: An ideas hatchery for future R development. In *Proceedings of the JSM*. AMSTAT, 1–9.

[115] Helen Shen et al. 2014. Interactive notebooks: Sharing the code. *Nature* 515, 7525 (2014), 151–152.

[116] Christopher Anthony Silles. 2014. *Provenance-aware CXXR*. Ph.D. Dissertation. University of Kent.

[117] Chris A. Silles and Andrew R. Runnalls. 2010. Provenance-awareness in R. In *Proceedings of the IPAW*. Springer, 64–72.

[118] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. 2005. A survey of data provenance in e-science. *SIGMOD Rec.* 34, 3 (2005), 31–36.

[119] Sébastien Sorlin and Christine Solnon. 2005. Reactive tabu search for measuring graph similarity. In *Proceedings of the IAPR*. Springer, 172–182.

[120] Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. 2014. Looking inside the black-box: Capturing data provenance using dynamic instrumentation. In *Proceedings of the IPAW*. Springer, 155–167.

[121] Jean-Luc Richard Stevens, Marco Elver, and James A. Bednar. 2013. An automated and reproducible workflow for running and analyzing neural simulations using Lancet and IPython Notebook. *Front. Neuroinform.* 7, 44 (2013), 44.

[122] Michael Stonebraker, Gerald Held, Eugene Wong, and Peter Kreps. 1976. The design and implementation of INGRES. *ACM Trans. Database Syst.* 1, 3 (1976), 189–222.

[123] Wang Chiew Tan et al. 2007. Provenance in databases: Past, current, and future. *IEEE Data Engineer. Bull.* 30, 4 (2007), 3–12.

[124] Dawood Tariq, Maisem Ali, and Ashish Gehani. 2012. Towards automated collection of application-level data provenance. In *Proceedings of the TaPP*. USENIX, 1–5.

[125] Håvar Valeur. 2005. *Tracking the Lineage of Arbitrary Processing Sequences*. Ph.D. Dissertation. Norwegian University of Science and Technology, Trondheim.

[126] André Van der Hoek. 2004. Design-time product line architectures for any-time variability. *Sci. Comput. Program.* 53, 3 (2004), 285–304.

[127] Jianwu Wang, Daniel Crawl, Shweta Purawat, Mai Nguyen, and Ilkay Altintas. 2015. Big data provenance: Challenges, state of the art and opportunities. In *Proceedings of the BigData*. IEEE, 2509–2516.

[128] Claes Wohlin. 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of International Conference on Evaluation and Assessment in Software Engineering*. ACM, 38:1–38:10.

[129] Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher et al. 2013. The Taverna workflow suite: Designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic Acids Res.* W557, 61 (2013), W557–W561.

[130] Zhaogui Xu, Ju Qian, Lin Chen, Zhifei Chen, and Baowen Xu. 2013. Static slicing for Python first-class objects. In *Proceedings of the QSIC*. IEEE, 117–124.

[131] Carlo Zaniolo. 1983. The database language GEM. *SIGMOD Rec.* 13, 4 (1983), 207–218.

[132] Yong Zhao, Mihael Hategan, Ben Clifford, Ian Foster, Gregor Von Laszewski, Veronika Nefedova, Ioan Raicu, Tiberiu Stef-Praun, and Michael Wilde. 2007. Swift: Fast, reliable, loosely coupled parallel computation. In *Proceedings of the SERVICES*. IEEE, 199–206.

[133] Yong Zhao and Shiyong Lu. 2008. A logic programming approach to scientific workflow provenance querying. In *Proceedings of the IPAW*. Springer, 31–44.

[134] Yong Zhao, Michael Wilde, and Ian Foster. 2006. Applying the virtual data provenance model. In *Proceedings of the IPAW*. Springer, 148–161.

Received September 2017; revised December 2018; accepted February 2019