

UNIVERSIDADE FEDERAL FLUMINENSE

**JOÃO FELIPE NICOLACI PIMENTEL**

**PROVENANCE FROM SCRIPTS**

NITERÓI

2021

UNIVERSIDADE FEDERAL FLUMINENSE

**JOÃO FELIPE NICOLACI PIMENTEL**

## **PROVENANCE FROM SCRIPTS**

Thesis presented to the Computing Graduate Program of the Universidade Federal Fluminense in fulfillment of the requirements for the Ph.D degree. Topic Area: Computer Science.

Advisor:

**VANESSA BRAGANHOLO MURTA**

Co-advisor:

**LEONARDO GRESTA PAULINO MURTA**

NITERÓI

2021

Ficha catalográfica automática - SDC/BEE  
Gerada com informações fornecidas pelo autor

P644p Pimentel, João Felipe Nicolaci  
Provenance from Scripts / João Felipe Nicolaci Pimentel ;  
Vanessa Braganholo, orientadora ; Leonardo Gresta Paulino  
Murta, coorientador. Niterói, 2021.  
223 f. : il.

Tese (doutorado)-Universidade Federal Fluminense, Niterói,  
2021.

DOI: <http://dx.doi.org/10.22409/PGC.2021.d.14709593728>

1. Proveniência. 2. Scripts. 3. Notebook interativo. 4.  
Reprodutibilidade de teste. 5. Produção intelectual. I.  
Braganholo, Vanessa, orientadora. II. Murta, Leonardo Gresta  
Paulino, coorientador. III. Universidade Federal Fluminense.  
Instituto de Computação. IV. Título.

CDD -

# JOÃO FELIPE NICOLACI PIMENTEL

## PROVENANCE FROM SCRIPTS

Thesis presented to the Computing Graduate Program of the Universidade Federal Fluminense in fulfillment of the requirements for the Ph.D degree. Topic Area: Computer Science.

Approved in April 2021.

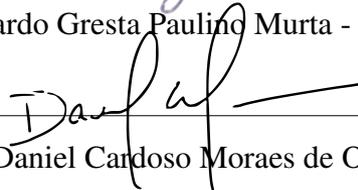
### APPROVED BY



Prof. D.Sc. Vanessa Braganholo Murta - Advisor / UFF



Prof. D.Sc. Leonardo Gresta Paulino Murta - Co-Advisor / UFF



Prof. D.Sc. Daniel Cardoso Moraes de Oliveira / UFF



Prof. D.Sc. Célio Vinicius Neves de Albuquerque / UFF



Prof. Ph.D. Juliana Freire de Lima e Silva / NYU



Prof. Ph.D. Paolo Missier / Newcastle University



Prof. D.Sc. Marta Lima de Queirós Mattoso / COPPE/UFRJ

Niterói

2021

# Acknowledgements

I want to thank my parents Angela Maria and Rui Alberto, for all the love, care, and support throughout my life. This work would not be possible without them.

My brothers Marcelo and Carlos Augusto, and my nephew Lucas for their friendship and distractions in stressful moments.

My advisors Leonardo Murta and Vanessa Braganholo, for the opportunity, guidance, and great advice. To Juliana Freire for her great contribution to this work since the beginning and for receiving me at NYU during my Ph.D. sandwich.

The collaborator Paolo Missier for his contributions in the Versioned-PROV extension. The collaborators Saumen Dey, Timothy McPhillips, Khalid Belhajjame, David Koop, and Bertram Ludäscher, for their contributions in combining noWorkflow and YesWorkflow. I would like to thank Vynicius Pontes for his contributions in using Git as a content database.

The collaborators in topics not related to the thesis for the opportunity of working with them and all the learning from these experiences: Vitor Lemos, Catarina Costa, Jose Jair Figueiredo, Anita Sarma, Henrique Linhares, Troy Kohwalter, Erica Mourão, Marcos Kalinowski, Emilia Mendes, Claes Wohlin, José William Menezes, Bruno Trindade, Tayane Moura, and Alexandre Plastino.

The YARG meetings, for all the valuable discussions, socialization, English practicing, and welcome environment. UFF, for the excellent academic support for learning and research in the last 11 years.

CNPq, for the Ph.D. scholarships (GM/GD 134435/2014-1 and GM/GD 160325/2015-3). Capes, for the sandwich scholarship (PDSE 88881.131563/2016-01).

# Resumo

Muitos cientistas usam scripts para projetar os experimentos visto que linguagens de script incorporam estruturas de dados sofisticadas, sintaxe simples e facilidade de obter resultados, sem a necessidade de investir tempo projetando sistemas. Cientistas podem escrever scripts em diversas ferramentas, como editores de texto, IDEs ou notebooks interativos. Editores de texto são leves e podem ser usados em qualquer máquina, mas não possuem muitos recursos para auxiliar o desenvolvimento. IDEs possuem recursos para melhorar a qualidade do desenvolvimento, mas possuem poucos recursos que auxiliam a análise e o desenvolvimento exploratório de experimentos. Notebooks permitem combinar códigos de scripts, textos, resultados de execuções e visualizações ricas, auxiliando no desenvolvimento exploratório e análise interativa de resultados. Entretanto, notebooks perdem em qualidade por conta de estados escondidos e células desordenadas, dificultando o entendimento e reprodutibilidade. Além desses problemas específicos de ferramentas, scripts falham em garantir a reprodutibilidade de experimentos e apresentam dificuldades no entendimento e gerenciamento de dados. Por exemplo, assuma que um cientista realize diversos ensaios (i.e., execuções de scripts de experimentos) com diferentes dados de entrada e obtenha uma grande quantidade de dados como resultado. Após a execução dos ensaios, ele precisará entender cada ensaio, relacionar resultados a dados de entrada e garantir a reprodutibilidade do experimento. Essas tarefas podem ser realizadas com a ajuda da proveniência dos scripts. Proveniência refere-se ao histórico de um objeto e todos os processos pelos quais ele passou em seu ciclo de vida. Contudo, a captura, gerência e análise de proveniência de scripts impõem diversos desafios. Primeiramente, é necessário decidir quais informações de proveniência são relevantes para compreensão e reprodutibilidade. Além disso, é necessário armazenar e compartilhar a proveniência coletada para permitir reproduções. Ainda, como vários ensaios são feitos no ciclo de vida de experimentos, é desejável obter o histórico de ensaios com diversas versões de proveniência. Finalmente, proveniência refere-se a diversos tipos de dados, que suportam diversas formas de análise com visualizações e consultas. Em notebooks, a proveniência também pode auxiliar a manter a ordem da execução e garantir o entendimento e qualidade de experimentos. Este trabalho tem quatro contribuições principais: um estudo do estado-da-prática do uso de scripts em experimentos, um estudo do estado-da-arte do uso de proveniência em scripts com uma proposta de taxonomia, concepção e implementação de ferramentas para capturar proveniência de scripts com o objetivo de auxiliar a reprodutibilidade e entendimento de experimentos, e ferramentas para capturar proveniência de notebooks interativos com o objetivo de auxiliar a reprodutibilidade e qualidade.

**Palavras-chave:** proveniência, scripts, notebooks, reprodutibilidade, qualidade, entendimento.

# Abstract

Many scientists use scripts for designing experiments, since script languages incorporate sophisticated data structures, simple syntax, and easiness to obtain results without spending much time on designing systems. Scientists write scripts in many tools, such as text editors, IDEs, or interactive notebooks. Text editors are lightweight and can be used in any machine, but they do not have many features to assist the development. IDEs have features to improve the quality of the scripts, but lack features to assist in experiment analyses and exploratory research. Notebooks combine script code, text, execution results and rich media, assisting in exploratory research and interactive analyses of results. However, notebooks lack in the quality of scripts due to hidden-states and unordered cells, hindering the understanding and reproducibility. Besides the issues associated with tools, scripts also fail to guarantee the reproducibility of experiments, and they present challenges for data management and understanding. For instance, assume that a scientist performs many trials (i.e., executions of experiment's scripts) with different input data and obtains a big amount of data as results. After executing these trials, she will need to understand each trial, relate results to input data, and guarantee the experiment reproducibility. Such tasks can be performed with the help of provenance, which refers to the history of an object and all processes it has been through in its life cycle. Nonetheless, collecting, managing, and analyzing provenance from scripts imposes diverse challenges. First, it requires deciding which script provenance information is relevant for comprehension and reproducibility. Second, after collecting provenance, it is necessary to store and share it to support reproducibility. Additionally, since many trials occur during the life cycle of experiments, it is desirable to capture the trial history as well, with multiple versions of provenance. Finally, provenance refers to a broad set of data types, which allows multiple forms of analysis, with visualizations and queries. In notebooks, provenance can also help in maintaining the execution order and assist with the understanding and quality of experiments. This thesis has four main contributions: a study of the state-of-the-practice usage of scripts in experiments, a study of the state-of-the-art usage of provenance in scripts with a taxonomy proposal, conception and implementation of tools to collect provenance from scripts aiming to assist their reproducibility and understanding, and tools to collect provenance from interactive notebooks aiming to assist their quality and reproducibility.

**Keywords:** provenance, scripts, notebooks, reproducibility, understanding, quality.

# List of Figures

1.1	Life cycle of scientific experiments [adapted from Mattoso et al. (2010)]. . . . .	1
1.2	Experiment provenance example. Ellipses represent entities. Rectangles represent activities. Meaning of labels: use – used; gen – wasGeneratedBy; der – wasDerivedFrom. . . . .	3
1.3	A mix of activities, data, and functions as first class objects. . . . .	6
2.1	Number of participants that answered each question. . . . .	13
2.2	(a) P1 - Education level (100 participants); (b) P2 - Number of experiments performed in computational environments (100 participants). . . . .	14
2.3	P3 - Scientific domains (100 participants). Answers starting with “O ” were not predefined in the questionnaire. . . . .	14
2.4	P4 - Expertise in years (99 participants). . . . .	15
2.5	P5 - Role of participants when they performed computational experiments (98 participants). Answers starting with “O ” were not predefined in the questionnaire. . . . .	15
2.6	P6 - Country of residence (89 participants). . . . .	15
2.7	RQ.Q1 – (a) preferred/more often used tools (95 participants) grouped by categories, (b) word cloud, and (c) Venn Diagram. . . . .	17
2.8	RQ.Q2 - favorite tool (92 participants). Answers starting with “O ” were not predefined in the questionnaire. . . . .	18
2.9	RQ.Q3 - reasons for tool preference (68 participants). . . . .	18
2.10	Distribution of Python constructs in scripts. This figure groups constructs into categories. The constructs of a category appear on the right of the category bar. A category corresponds to the union of its constructs. . . . .	24

---

2.11	Regions of scripts. Numbers in region descriptions refer to the number of scripts that have the feature in the specified region unless stated otherwise. Note that a script may have multiple features in the same region. . . . .	26
2.12	Modules – (a) Top 20 used modules, (b) All modules grouped by domains, and (c) All modules grouped by definition type. . . . .	29
2.13	Frequent module domains used together. . . . .	31
2.14	Strategies for processing data: (a) during input, (b) during output, (c) in the middle, (d) interweaving both input and output. . . . .	33
3.1	An example of an executed notebook with Markdown, code, and output. . . . .	39
3.2	Original notebook and two executions that follow different orders. . . . .	43
3.3	Three types of Hidden States: (a) Re-execution; (b) edited cell; (c) removed cell. . . . .	43
3.4	Top 15 most declared programming languages. Notebooks axis in logarithmic scale. . . . .	50
3.5	Distribution of code cells and maximum execution counter for overall group (a) and popular group (b). . . . .	51
3.6	Notebook corpus and its partitions used in the analyses. . . . .	52
3.7	Snippet of pythoncode/improvedlm.ipynb from the GitHub repository poorbaby/Predict-New-York-Taxi-Demand. . . . .	55
3.8	Distribution of code cells in executed notebooks (a) and popular notebooks (b). . . . .	56
3.9	Distribution of skips in notebooks with unambiguous execution order (a) and popular notebooks (b). . . . .	57
3.10	Snippet of pparker-roach/project_7-SANDBOX.ipynb from the GitHub repository mohsseha/DSI-BOS-students. . . . .	58
3.11	Failure reasons for the executions in each execution mode. The blue bars represent the Top 10 exceptions. The “Timeout” orange bar represents executions that we stopped when they took 5 minutes to run. The “Other” orange bar groups all the other exceptions that are not part of the Top 10. . . . .	63
4.1	Snowballing provenance. . . . .	78
4.2	Selected papers in Snowballing. . . . .	78

---

4.3	Distribution of work by publishing location. . . . .	81
4.4	Main taxonomy of provenance from scripts. . . . .	81
4.5	Toy experiment that classifies a yearly precipitation data from Rio de Janeiro. . . . .	82
4.6	Provenance classification systems. . . . .	84
4.7	Expanded <i>Collection</i> taxonomy node of Figure 4.4. . . . .	84
4.8	Observed and disclosed strategies. . . . .	88
4.9	Expanded <i>Management</i> taxonomy node of Figure 4.4. . . . .	90
4.10	Expanded <i>Analysis</i> taxonomy node of Figure 4.4. . . . .	95
5.1	Intentionally simple implementation of the happy numbers problem. . . . .	115
5.2	Transformed script with function definitions and function call. . . . .	116
5.3	noWorkflow 1 relational data model. Green represent additions. . . . .	122
5.4	noWorkflow 2 relational data model. Green represent additions and semantic changes. Purple represents renames. It does not show removals. . . . .	124
5.5	Subset of Prolog facts from a trial. We reordered lines and added line breaks when needed to fit the page. . . . .	127
5.6	Version model example. . . . .	130
5.7	Evolution history. Nodes represent trial versions . . . . .	131
5.8	SQL query. . . . .	135
5.9	Prolog query. . . . .	135
5.10	Command that shows the activations of Trial 1. . . . .	136
5.11	ORM query. . . . .	137
5.12	Python pattern matching query. . . . .	137
5.13	<code>now vis</code> web page. . . . .	138
5.14	Dataflow graph. . . . .	140
5.15	Snippet of brief diff command. . . . .	141
5.16	Comparison of activation graphs. . . . .	142
6.1	Provenance collection in notebook using noWorkflow extension. . . . .	152

---

6.2	Provenance analysis in a notebook. . . . .	154
6.3	Notebook cleaning using provenance. . . . .	158
6.4	Julynter in action (left pane). By analyzing the notebook on the right pane, Julynter identified ten issues from four different categories. . . . .	159
6.5	Architecture of Julynter. Blue arrows represent input messages that occur before the cell execution. Red arrows represent output messages that occur after the kernel executes the cell. . . . .	162
6.6	Participants experiment flow. . . . .	164
6.7	Participants' experience. . . . .	164
6.8	Solved and unsolved lints. . . . .	166
6.9	Satisfaction with the lint groups. . . . .	168
6.10	Chosen words in the Microsoft Product Reaction Cards (BENEDEK; MINER, 2002). The colors vary according to the experiment phase in a gradient. Mixed colors indicate that participants of both phases chose the word and the mixing intensity indicates the proportion. . . . .	169
A.1	<i>Floyd-Warshall</i> implementation (A) and encoded input graph (B). . . . .	205
A.2	Plain PROV mapping of $dist_i[j] = ikj$ . . . . .	208
A.3	PROV-Dictionary mapping of $dist_i[j] = ikj$ . . . . .	209
A.4	Versioned-PROV mapping of $dist_i[j] = ikj$ . . . . .	210
A.5	Number of PROV, PROV-Dictionary, and Versioned-PROV PROV-N statements for list definitions, reference derivations, and part assignments (A) and total number of statements (B). . . . .	213
A.6	Overhead functions of part assignments. . . . .	214
A.7	Overhead functions for list definitions (A) and derivations by reference (B). . . . .	214
B.1	Activations that produce a failing value frequently. . . . .	217
A.1	Size of content database directories of all script executions for each content database type [adapted from Pontes (2018)]. . . . .	220
A.2	Duration of all script executions for each content database type version [adapted from Pontes (2018)]. . . . .	222

# List of Tables

2.1	Association Rules for Module Usage. Domains abbreviated as follows: <i>I/O</i> – Input/Output; <i>DS</i> – Data Structure; <i>Pattern</i> – Pattern Recognition; <i>Vis</i> – Visualization. . . . .	30
2.2	Ways of defining paths for input and output files. The Input and Output columns indicate the number of scripts that apply these ways for input and output files, respectively. . . . .	32
3.1	Execution modes for the reproducibility experiments. . . . .	44
3.2	Normalization Operations for Comparing Execution Results. . . . .	46
3.3	Results of research questions related to prospective data. . . . .	53
3.4	Output formats in cells and notebooks. Note that a cell can have multiple output formats, thus, the percentages add up to more than 100%. . . . .	54
3.5	Association rules related to timeout . . . . .	62
3.6	Association rules related to skips and <i>NameError</i> . . . . .	64
3.7	Reproducibility results for all notebooks. . . . .	66
3.8	Reproducibility results for the popular group. . . . .	67
3.9	Association rules related to executions that generate the same results after the execution counter normalization. . . . .	68
4.1	Selected approaches with provenance support: main and secondary goals. Labels in secondary goals column refer to goals: <i>Cache</i> – Caching; <i>Compr</i> – Comprehension; <i>Frame</i> —Framework; <i>Manag</i> – Management; <i>Repro</i> – Reproducibility. . . . .	79
4.2	Selected approaches in the update with provenance support: main and secondary goals. Labels in secondary goals column refer to goals: <i>Cache</i> – Caching; <i>Compr</i> – Comprehension; <i>Frame</i> —Framework; <i>Manag</i> – Management; <i>Repro</i> – Reproducibility. . . . .	103

5.1	Provenance collection strategies. Labels in Annotations columns refer to categories described in Chapter 4 <i>Exte</i> —External; <i>Inte</i> —Internal; <i>Pars</i> —Parseable; <i>Exec</i> —Executable; <i>Incl</i> —Inclusive; <i>Excl</i> —Exclusive; <i>Defi</i> —Definition; <i>Prov</i> —Provenance; <i>Man</i> —Mandatory; <i>Opt</i> —Optional. . . . .	118
5.2	Provenance management classification. . . . .	133
5.3	Provenance analysis classification, based on Query, Visualization, and Diff. . .	143
6.1	Issues detected by Julynter. The first character of the Code indicates the category: C – Confuse Notebook; H – Hidden State; I – Import; P – Path; T – Invalid Title . . . . .	161
6.2	Julynter usage statistics. . . . .	165
A.1	Versioned-PROV types. . . . .	206
A.2	Versioned-PROV attributes. . . . .	207
A.1	Average sizes of the content database after 4 executions for each content database type and reduction percentage between the Git content database and the baseline [adapted from Pontes (2018)]. . . . .	220
A.2	Size in Megabytes of content database directory for each noWorkflow trial using the synthetic script and the execution of <code>now gc</code> at the end [adapted from Pontes (2018)]. . . . .	221
A.3	Average execution duration after 4 executions for each content database and average differences between Git content database and baseline [adapted from Pontes (2018)]. . . . .	222

# List of Acronyms and Abbreviations

adapr	:	Accountable Data Analysis Process in R;
API	:	Application Programming Interface;
AST	:	Abstract Syntax Tree;
DAG	:	Directed Acyclic Graph;
DB-BRAS	:	Mailing list of Database Researchers in Brazil;
CoRR	:	Cloud of Reproducible Records;
CPL	:	Core Provenance Library;
DDG	:	Data Derivation Graph;
ESSW	:	Earth System Science Workbench;
HTML	:	HyperText Markup Language;
IDE	:	Integrated Development Environment;
IDL	:	Interactive Data Language;
INRIA	:	Institut National de Recherche en Informatique et en Automatique;
JSON	:	JavaScript Object Notation;
LCS	:	Longest Common Subsequence;
LNCC	:	Laboratório Nacional de Computação Científica;
MATLAB	:	Matrix Laboratory;
NIST	:	National Institute of Standards and Technology;
NYU	:	New York University;
OPM	:	Open Provenance Model;
ORM	:	Object Relational Mapper;
OS	:	Operating System;
PDF	:	Portable Document Format;
PEP	:	Python Enhancement Proposal;
PNG	:	Portable Network Graphics;
POSIX	:	Portable Operating System Interface;
REST	:	Representational State Transfer;
RQ	:	Research Question;
SMLD	:	Secure Machine Learning Debugger;

---

SQL	:	Structured Query Language;
SVG	:	Scalable Vector Graphics;
SWfMS	:	Scientific Workflow Management System;
Tcl	:	Tool command language;
UFF	:	Universidade Federal Fluminense;
UFRJ	:	Universidade Federal do Rio de Janeiro;
UPenn	:	University of Pennsylvania;
USP	:	Universidade de São Paulo;
UTF	:	Unicode Transformation Format;
VCS	:	Version Control System;

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Context . . . . .	1
1.2	Motivation . . . . .	4
1.3	Problem . . . . .	5
1.4	Hypothesis and Goals . . . . .	8
1.5	Organization . . . . .	9
<b>2</b>	<b>State-of-the-Practice: Scripts</b>	<b>11</b>
2.1	Introduction . . . . .	11
2.2	Questionnaire . . . . .	11
2.2.1	Materials and Methods . . . . .	12
2.2.1.1	Distribution . . . . .	12
2.2.1.2	Population . . . . .	13
2.2.2	Results . . . . .	16
2.2.2.1	RQ.Q1. What are the scientists' preferred/more often used tools to run experiments? . . . . .	16
2.2.2.2	RQ.Q2. Which tool is their favorite? . . . . .	16
2.2.2.3	RQ.Q3. What are the reasons for your preference? . . . . .	17
2.2.3	Threats to Validity . . . . .	19
2.3	Script Analysis . . . . .	20
2.3.1	Materials and Methods . . . . .	21
2.3.1.1	Analyses . . . . .	21

---

2.3.1.2	Data Collection . . . . .	23
2.3.2	Results . . . . .	24
2.3.2.1	RQ.S1. How do scientists structure scripts? . . . . .	24
2.3.2.2	RQ.S2. How do scientists use modules and external tools? . . . . .	27
2.3.2.3	RQ.S3. How do scientists process data in scripts? . . . . .	31
2.3.3	Threats to Validity . . . . .	34
2.4	Discussion . . . . .	35
<b>3</b>	<b>State-of-the-Practice: Notebooks</b>	<b>36</b>
3.1	Introduction . . . . .	36
3.2	Background . . . . .	38
3.3	Materials and Methods . . . . .	40
3.3.1	Research Questions and Analyses . . . . .	40
3.3.2	Data Acquisition and Preprocessing . . . . .	47
3.3.3	Popular Notebooks Selection . . . . .	48
3.3.4	Sampling . . . . .	48
3.3.5	Corpus . . . . .	49
3.4	Results . . . . .	52
3.4.1	RQ.N5. Do users store notebooks with retrospective data? . . . . .	52
3.4.2	RQ.N6. How are notebooks executed? . . . . .	56
3.4.3	RQ.N7. How reproducible are notebooks? . . . . .	59
3.5	Threats to Validity . . . . .	69
3.6	Discussion . . . . .	72
<b>4</b>	<b>State-of-the-Art on Provenance from Scripts</b>	<b>73</b>
4.1	Introduction . . . . .	73
4.2	Related Work . . . . .	74

---

4.3	Taxonomy . . . . .	80
4.3.1	Provenance Collection . . . . .	82
4.3.1.1	Annotations . . . . .	84
4.3.1.2	Definition Provenance . . . . .	85
4.3.1.3	Deployment Provenance . . . . .	86
4.3.1.4	Execution Provenance . . . . .	87
4.3.2	Provenance Management . . . . .	90
4.3.2.1	Storage . . . . .	90
4.3.2.2	Sharing . . . . .	92
4.3.2.3	Reproducibility . . . . .	93
4.3.2.4	Versioning . . . . .	94
4.3.3	Provenance Analysis . . . . .	95
4.3.3.1	Query . . . . .	95
4.3.3.2	Visualization . . . . .	96
4.3.3.3	Comparison . . . . .	98
4.3.4	Applicability to Other Provenance Systems . . . . .	98
4.4	Threats to Validity . . . . .	100
4.5	Update . . . . .	101
4.6	Discussion . . . . .	105
<b>5</b>	<b>Provenance in Scripts</b>	<b>108</b>
5.1	Introduction . . . . .	108
5.2	Provenance Collection . . . . .	110
5.2.1	Definition Provenance . . . . .	110
5.2.2	Deployment Provenance . . . . .	112
5.2.3	Execution Provenance . . . . .	113
5.2.4	Summary . . . . .	118

---

5.3	Provenance Management . . . . .	121
5.3.1	Storage . . . . .	122
5.3.2	Sharing . . . . .	126
5.3.3	Reproducibility . . . . .	127
5.3.4	Versioning . . . . .	128
5.3.5	Summary . . . . .	132
5.4	Provenance Analysis . . . . .	134
5.4.1	Query . . . . .	134
5.4.2	Visualization . . . . .	138
5.4.3	Comparison . . . . .	140
5.4.4	Summary . . . . .	142
5.5	Limitations . . . . .	144
5.6	Discussion . . . . .	147
<b>6</b>	<b>Provenance in Notebooks</b>	<b>148</b>
6.1	Introduction . . . . .	148
6.2	Best Practices . . . . .	149
6.3	noWorkflow for Notebooks . . . . .	150
6.3.1	Extension . . . . .	150
6.3.1.1	Collection . . . . .	151
6.3.1.2	Analysis . . . . .	152
6.3.2	Kernel . . . . .	155
6.3.2.1	Collection . . . . .	155
6.3.2.2	Cleaning . . . . .	156
6.4	Julynter . . . . .	158
6.4.1	Approach . . . . .	159
6.4.2	Experiment Design . . . . .	162

---

6.4.3	Data Collection . . . . .	163
6.4.4	Results and Discussion . . . . .	165
6.4.5	Threats to Validity . . . . .	171
6.5	Discussion . . . . .	172
<b>7</b>	<b>Conclusion</b>	<b>173</b>
7.1	Contributions . . . . .	173
7.2	Future Work . . . . .	175
7.3	Publications and Awards . . . . .	177
	<b>References</b>	<b>179</b>
	<b>Appendix A - Versioned-PROV</b>	<b>203</b>
A.1	Introduction . . . . .	203
A.2	Running Example . . . . .	205
A.3	Versioned-PROV . . . . .	206
A.3.1	Concepts . . . . .	206
A.3.2	Mapping Example . . . . .	207
A.4	Evaluation . . . . .	211
A.5	Final Remarks . . . . .	214
	<b>Appendix B - Version Model Evaluation</b>	<b>216</b>
	<b>Annex A - Git Integration Evaluation</b>	<b>218</b>
A.1	Introduction . . . . .	218
A.1.1	Materials and Methods . . . . .	218
A.1.1.1	Research Questions . . . . .	218
A.1.1.2	Corpus . . . . .	219
A.1.2	Results . . . . .	220

---

A.1.2.1	RQ.G1. Is there any reduction in the size of the content database? . . . . .	220
A.1.2.2	RQ.G2. Is there any performance overhead with the integration? . . . . .	222
A.2	Conclusion . . . . .	223

# Chapter 1

## Introduction

### 1.1 Context

In the past decades, scientists started to run experiments *in silico* for reducing their costs, increasing their productivity, and understanding the functioning of complex systems (HOBAN; BERTORELLE; GAGGIOTTI, 2012; TRAVASSOS; BARROS, 2003). Many experiments use many existing computational tools to process data and obtain results. Thus, scientists need to orchestrate these tools according to their necessities (HOBAN; BERTORELLE; GAGGIOTTI, 2012).

Mattoso et al. (2010) define the life cycle of scientific experiments as a loop composed of three main phases: composition, execution, and analysis (Figure 1.1). Each phase has a sub-cycle of activities. During the life cycle of experiments, scientists navigate multiple times through the sub-cycles and through the main cycle.

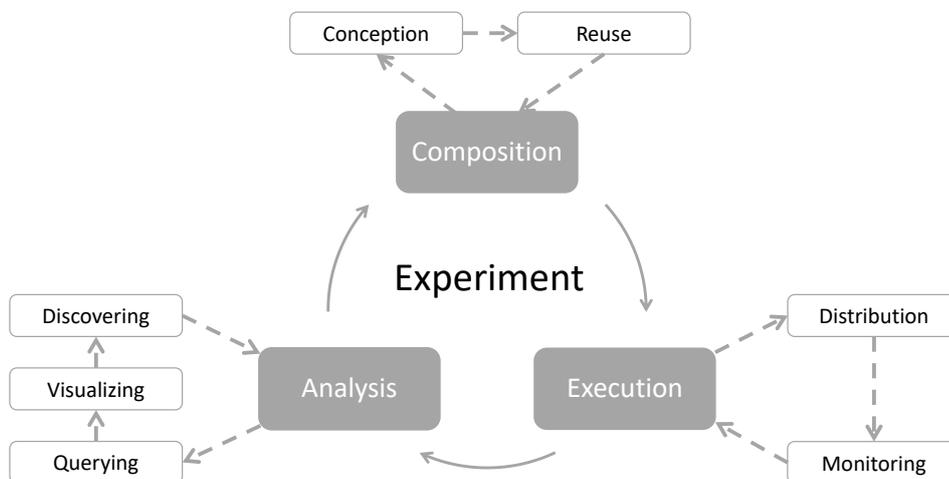


Figure 1.1: Life cycle of scientific experiments [adapted from Mattoso et al. (2010)].

During the *composition* phase, scientists formulate hypotheses and compose execution plans that enact the programs and data involved in the experiment. During the *execution* phase, they run the execution plans over input data, which represent a specific context or population for the experiment. Each execution of a computational experiment is called a *trial*. Finally, during the *analysis* phase, scientists query and visualize the results, seeking to elaborate conclusions to confirm or refute the hypotheses of the experiment. Depending on the analysis result, the scientists repeat the cycle using different input data and parameters, or refining the execution plan with the obtained knowledge.

For composing execution plans, scientists can use system programming languages (e.g., C, C++, Fortran, or Java), scripts (e.g., JavaScript, Perl, Python, or R), or Scientific Workflow Management Systems (SWfMS – e.g., Kepler, SciCumulus, Taverna, or VisTrails). Each one of these platforms presents advantages and disadvantages, as discussed in the following.

Ousterhout (1998) envisioned the importance of scripting languages for the 21<sup>st</sup> century. He indicated that the usage of scripts for gluing components could result in applications developed 5 to 10 times faster than using system programming languages. However, system programming languages would not cease to exist, because components written in these languages could run 10 to 20 times faster than with scripts.

Loui (2008) vindicated Ousterhout's vision by discussing the growth of script usage and identifying the power of scripts. Due to the lack of type declaration and structural requirements of system programming languages, scripts usually result in short and more expressive source code. Short source code allows programmers to turn ideas into code quickly, supporting rapid development and rapid prototyping. Additionally, scripts provide a kind of high-level programming that shields programmers against concerns related to performance and memory management. Finally, the flexibility of scripts is well suited for working with heterogeneous data and performing data transformations.

The power of scripts in gluing components and dealing with heterogeneous data motivated their use by the scientific community. Dubois (1999) advocates using scripting languages such as IDL, Matlab, Perl, Python, and Tcl for scientific programming instead of compiled programs. According to Dubois (1999), these scripting languages incorporate sophisticated data structures and give immediate feedback on algorithms. Similarly, Langtangen (2006) identifies the growth of script usage in scientific experiments because of their simple syntax, easiness to visualize results, and easiness to combine different tools. Jackson (2002) states the importance of Python for applications in science and engineering, due its support for high-level object-oriented programming with automatic memory management, dynamic typing and binding in an

easy-to-learn syntax, its wide variety of built-in data structures and algorithms, and its easiness to integrate to native C/C++/Fortran code, among other reasons.

While many people have been using and advocating the usage of scripts for scientific experiments, scientific workflow management systems (SWfMS) were proposed with a similar goal of gluing components and supporting scientists (ALTINTAS; BARNEY; JAEGER-FRANK, 2006; BOWERS; MCPHILLIPS, T. M.; LUDÄSCHER, 2008; CALLAHAN et al., 2006; KIM et al., 2008; OLIVEIRA et al., 2010; SIMMHAN; PLALE; GANNON, 2008; WOLSTENCROFT et al., 2013; ZHAO et al., 2007). Scientific workflows are essentially directed acyclic graphs (DAG) representing a computation (CHENEY; AHMED; ACAR, 2011). They often implement execution engines that schedule and distribute parallel execution to environments like clusters, clouds, and HPC. Some SWfMS have huge communities and impact: Galaxy has a huge community of bioinformatics users; Pegasus has important results with complex workflows, such as the awarded Ligo experiment (RAMAKRISHNAN et al., 2007).

Some SWfMS, such as Swift/T (ZHAO et al., 2007), dispel4py (FILGUIERA et al., 2017), and Snakemake (KÖSTER; RAHMANN, 2012), propose scripting languages for defining the workflow but restrict the language to a syntax that supports the creation of a DAG. Different from general-purpose scripts, many SWfMS not only support gluing components together but also collect their provenance.

According to Moreau et al. (2008b), provenance refers to the documented history of processes in the life cycle of a computational object. Figure 1.2 presents a simple experiment provenance. This experiment was composed of two activities (i.e., processes): `simulation` and `plot`. The `simulation` activity used the entities `data1.dat` and `data2.dat` for generating the entity `result`. Hence, the provenance suggests that `result` was derived from both `data1.dat` and `data2.dat`. Similarly, `plot` used `result` and generates `figure.svg`. It is possible to navigate the provenance graph to find the documented activities and entities that contributed to the generation of `figure.svg`.

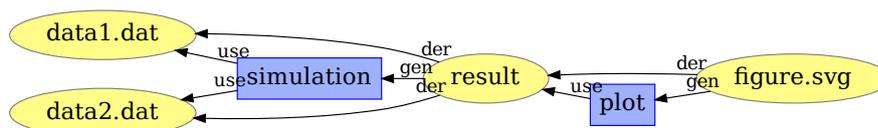


Figure 1.2: Experiment provenance example. Ellipses represent entities. Rectangles represent activities. Meaning of labels: use – used; gen – wasGeneratedBy; der – wasDerivedFrom.

## 1.2 Motivation

Provenance plays an important role in scientific experiments. Many analysis results motivate repetitions in the life cycle of experiments. 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 cycle 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 thinking in the cycles and take advantage of them, by using cycles to alternate the input data and some experimental activities. For instance, this occurs in simulations with parameter sweeping (WALKER; GUIANG, 2007; DIAS, J. et al., 2011). In these simulations, each iteration deals with a combination of parameters. In all the situations that motivate the repetition of the cycle, the knowledge is cumulative, and scientists can use data from previous trials in further analyses.

When scientists compare data from different trials, some questions arise: (i) “Which parts of the input data influence a given result in a trial?” and (ii) “How was the execution flow of each trial?”. Provenance can help to answer these questions. In the context of scientific experiments, the provenance considers not only input and output data, but also environment characteristics, processes applied to input data for transforming it into output data, intermediate data of these processes, and other contextual metadata (e.g., duration, usage of computational resources) of each process and of the experiment itself.

Besides helping scientists to know which of the many trial-and-error paths produced a particular result, where the result came from, and which process led to the result (SILVA; TOHLIN, 2008), provenance has many other applications. It enables sharing experiment results with computation and input data (KOSLOW, 2002), allowing others to replicate them (MILES et al., 2007). Scientists can use provenance to check integrity and authenticity of experiments (LYNCH, 2000), check data quality, audit, and understand experiments (SIMMHAN; PLALE; GANNON, 2005b). Besides the usage in scientific experiments, provenance also supports detecting system dependencies (CHIRIGATI et al., 2016), detecting intrusion (MARTIN; LYLE; NAMILUKO, 2012), debugging (LINHARES et al., 2019), detecting system changes (MUNISWAMY-REDDY et al., 2006), and can be used in education (DAVIDSON; FREIRE, 2008).

Despite the ability to define experiments and the support for provenance in SWfMS, many scientists still prefer to use scripts and notebooks in their experiments when they do not have to parallelize executions or use existing SWfMS components. Some even say that most scientists

still use scripts instead of SWfMS due to their flexibility in customizations (DEY et al., 2015). Additionally, some initiatives (e.g., Software Carpentry<sup>1</sup>) use scripts and interactive notebooks to teach computing skills for researchers. However, when using scripts or interactive notebooks, those scientists lose the ability to collect provenance and, consequently, all its benefits.

## 1.3 Problem

Collecting provenance without an SWfMS is challenging. While SWfMS provide well-defined and separate concepts of activities and data, enforcing their structuring as a DAG in a controlled environment, scripts can contain a chaotic combination of control flows, cycles, and functions as first-class objects (STRACHEY, 2000), in an uncontrolled environment. These differences create challenges for provenance collection in scripts (MURTA et al., 2014), as presented next.

First, the **presence of control flow and cycles in scripts** makes it hard to identify which functions contributed to the generation of a given data product. This occurs because the only identifications of a function are its name and its definition, but scripts can call a single function multiple times with different parameters during the program execution. On the other hand, the DAG structure of workflow makes it easier to identify activities uniquely.

Second, the **lack of well-defined concepts for activities and data** in scripts makes it hard to determine what should be captured and at what level of granularity. If we collect coarse-grained provenance, such as the script or process as an activity, and its arguments as data, users may lose important details of the script (TARIQ; ALI; GEHANI, 2012). In contrast, if we collect very fine-grained provenance, such as expressions as activities, and variables as data, users may get overwhelmed with a large volume of data to analyze. Even when using an intermediate granularity of provenance, such as function calls as activities and parameters as data, users may have difficulties to identify what is an activity and what is data. For instance, in Figure 1.3 we call `polish_eval` passing a `deque` of operations and numbers in polish notation. When the value at the beginning of the `deque` is a number, the function `polish_eval` just returns it, as a data, but when the value is an operation, the function evaluates it as if it were an activity. Note that the effect of this code is composing an invocation tree as follows: `add(sub(2, 1), add(3, 4))`, which evaluates to `add(1, 7)`, and then to 8.

Another challenge in collecting provenance from scripts is **determining who should do the collection**. Letting the users instrument their scripts to collect the provenance by themselves imposes a lot of effort, which is not desirable nor appreciated (HUQ; APERS; WOMBACHER,

---

<sup>1</sup><http://software-carpentry.org/>

```
1 from operator import add, sub
2 from collections import deque
3
4 def polish_eval(operation):
5     first = operation.popleft()
6     if isinstance(first, int):
7         return first
8     return first(polish_eval(operation), polish_eval(operation))
9
10 op = deque([add, sub, 2, 1, add, 3, 4])
11 print(op)
12 print(polish_eval(op))
```

Figure 1.3: A mix of activities, data, and functions as first class objects.

2013b). Additionally, it is error-prone since the instrumentation defined by the user may not represent the script definition after the evolution of experiments in the exploratory analysis. However, automatic collection imposes scalability challenges in scripts. Does the user want to know what happens inside `polish_eval` in Figure 1.3? Does she just want to know that the printed result (i.e., 8) is the result of `polish_eval` applied to `op`? Collecting provenance with the full depth of function calls when the user just wants a shallow vision of the execution may result in unnecessary overhead and impair the usage of the provenance (STAMATOGLIANNAKIS; GROTH; BOS, 2014).

Additionally, depending on the way an approach handles automatic collection, it may introduce **false positives and false negatives** (TARIQ; ALI; GEHANI, 2012). For instance, if it uses temporal information to collect provenance in Figure 1.3, it may say that the `polish_eval` call in line 12 depends on the `print` call in line 11, which is a false positive since both calls are independent. On the other hand, if it does not look at what happens inside the recursive calls to `polish_eval`, it may not identify that the second `polish_eval(operation)` in line 8 depends on the first one due its `popleft` operations in line 5. This would represent a false negative.

Moreover, **scripts run outside of controlled environments**. Different from SWfMS, which bundles a set of tools to manage the workflow executions, one cannot make assumptions beyond the existence of the script interpreter. Thus, one cannot assume that there is a version of the script in a version control system, or that a user is executing a script with the very same dependencies and environment variables the developer used to implement it.

Scientists use different types of tools to write scripts, such as text editors, IDEs, and interactive notebooks. Each one of these tools present advantages and disadvantages. Text editors are simple and generic tools that work with most programming languages. Usually, they are lightweight and can run at most machines, from personal computers to remote servers. While these editors are good enough for writing code, they lack features to improve the development

quality. IDEs build on top of text editors and fill this gap by introducing features to support development tasks and ensuring quality for specific programming languages. However, despite supporting the development of scripts by programmers, most IDEs lack features to support scientists in experiment analyses and exploratory research. For this reason, many scientists use interactive notebooks to assist in these tasks. In fact, the traffic to the Jupyter Notebook website suggests that more than 500,000 people actively use it (SHEN et al., 2014).

Interactive notebooks are computational environments based on literate programming (KNUTH, 1984) that allow users to write documents containing script code, text, plots, and other rich media. Users can use interactive notebooks to run computations and visualize their results interactively. Users can share interactive notebooks and convert them into other formats, such as HTML or PDF. Two well-known interactive notebook environments are Jupyter Notebook<sup>2</sup> and knitr<sup>3</sup>. Kluyver et al. (2016) advocate the usage of notebooks<sup>4</sup> for publishing reproducible research due to their ability to combine reporting text with the executable research code.

However, the format of interactive notebooks has been increasingly criticized for **encouraging bad habits that lead to unexpected behavior and are not conducive to reproducibility** (POMOGAJKO, 2015; GRUS, 2018; MUELLER, 2018; PIMENTEL et al., 2019b). Among the main criticisms are hidden states, unexpected execution order with fragmented code, and bad practices in naming, versioning, testing, and modularizing code. In addition, the notebook format does not encode library dependencies with pinned versions, making it difficult (and sometimes impossible) to reproduce the notebook. These criticisms reinforce prior work, which has emphasized the negative impact of the lack of Software Engineering best practices in scientific computing software (WILSON et al., 2014), regarding separation of concerns (HÜRSCH; LOPES, 1995), tests (MYERS et al., 2004), and maintenance (HORWITZ; REPS, 1992). Finally, since interactive notebooks use scripts, they also face the challenges of collecting provenance without an SWfMs. However, they have the additional challenge of unordered execution. Cells in interactive notebooks can be executed at any other and each execution order may lead to different results.

Due to the specific challenges of collecting provenance from scripts, **analyzing and managing the collected provenance also impose distinct challenges**. First, analyzing graphs from scripts require summarizing and querying techniques that consider different aspects of scripts, such as cycles. Second, reproducing a script requires a storage and restore system that not only

---

<sup>2</sup><http://jupyter.org/>

<sup>3</sup><http://yihui.name/knitr/>

<sup>4</sup>We use the terms “interactive notebooks” and “notebooks” interchangeably throughout this work

considers the script definition, but also the used libraries and input data. Finally, scripts evolve over time. Managing and analyzing the evolution require techniques to deal with the storage overhead and the comparison of complex graphs.

## 1.4 Hypothesis and Goals

Given the aforementioned motivation, the hypothesis of this work is that scripts and interactive notebooks can also be supported by an infrastructure for collecting, managing, and analyzing provenance from experiments. More specifically, we target scientific scripts whose main purpose is gluing components but that also contain control flows, loops, other complex programming features. Hence, this work has four main goals:

1. Understand how scientists use scripts and interactive notebooks (state-of-the-practice assessment). For this goal, we ran a questionnaire with 120 scientists, identifying their favorite tools for running experiments and their reasoning. We also analyzed 172 experiments from a scientific repository to understand how they use scripts in practice. Finally, we analyzed 1.4 million notebooks from GitHub, extracting characteristics that impact their quality and reproducibility. We observed that scripts are widely used with almost no evidence of provenance usage.
2. Identify approaches that have been proposed to support provenance from scripts (state-of-the-art assessment). For this goal, we performed a systematic literature review through snowballing, in which we visited 1,345 references, and we ended up with 53 papers referring to 27 different approaches. Based on the snowballing results, we proposed a taxonomy for provenance from scripts. Among the existing related work, we noticed that very few approaches collect fine-grained provenance that includes variable dependencies, and none of them work with Python, which is a highly used language by scientists, and, except for approaches that rely on Git, no approaches provide mechanisms for analyzing the evolution of the collected provenance and comparing trials.
3. Conceive and implement a tool (noWorkflow<sup>5</sup>) that collects, manages, and analyzes provenance from scripts for supporting understanding and reproducibility. Regarding the collection, the approach requires no changes in the users' scripts, and it does not change the script's outcome. Additionally, it collects provenance at fine-grain and supports specifying the collection depth for avoiding the collection of data that is not interesting for the

---

<sup>5</sup><https://github.com/gems-uff/noworkflow>

user, such as the intermediate results of simple `add` operations in a function that returns the `sum` of all elements of a collection. Thus, it collects provenance from scripts in a transparent and automatic way, with configurable granularities to meet the expectations of different users.

For management, the approach provides mechanisms for working with multiple versions of provenance that appear in the life cycle of experiments. For efficiency, it applies techniques to reduce the storage overhead.

For analysis, the approach provides distinct querying and visualization mechanisms that allow users to find the desired results.

4. Conceive and implement tools (noWorkflow extension and kernel, Julynter<sup>6</sup>) that collect and analyze provenance from interactive notebooks for supporting quality and reproducibility. For this goal, we adapt noWorkflow to collect and analyze provenance on Jupyter Notebooks and to use the provenance for cleaning the notebooks. We also propose a set of best practices for working on notebooks and propose a standalone tool (Julynter) to assist in the application of these best practices. We evaluated Julynter in a remote experiment with users to assess its recommendations, usability, and improved it accordingly.

Thus, the main contributions of this thesis reside in providing an understanding of how scientific scripts and notebooks are used; and proposing an infrastructure for collecting and using provenance in scripts and notebooks.

## 1.5 Organization

Besides this introduction, this document is organized into six chapters.

Chapter 2 studies the state-of-the-practice on the usage of scripts for experiments. It attempts to understand how scientists use scripts. Similarly, Chapter 3 studies the state-of-the-practice on the usage of notebooks.

Chapter 4 presents the state-of-the-art of provenance from scripts. It categorizes provenance into a taxonomy and presents the available techniques for collecting each type of provenance. After presenting how to capture provenance, it presents techniques for storing and versioning provenance and using provenance for reproducibility. Finally, the chapter concludes by describing provenance querying and visualization.

---

<sup>6</sup><https://github.com/dew-uff/julynter>

---

Chapter 5 discusses provenance from scripts and proposes extensions for a tool that collects provenance from scripts (noWorkflow), introducing many features aligned with our aforementioned goals. For provenance collection, we propose performing fine-grained provenance collection, in addition to the existent coarse-grained collection (MURTA et al., 2014). For provenance management, we propose reducing the storage overhead, providing versioning and using provenance for reproducibility. Finally, for provenance analysis, we propose visualizing the fine-grained provenance, filtering it and querying it, visualizing the provenance evolution, and comparing trials.

Chapter 6 discusses provenance from notebooks. It describes the integration of noWorkflow to Jupyter Notebooks, proposes best practices for ensuring the quality and reproducibility of notebooks, and conceives and implements a new tool (Julynter) to assist the application of these best practices.

Finally, Chapter 7 concludes this document presenting the contributions, future work, and publications.

# Chapter 2

## State-of-the-Practice: Scripts

### 2.1 Introduction

In Chapter 1, we introduced that scientists use scripts for designing experiments since script languages incorporate sophisticated data structures, simple syntax, and easiness to obtain results without spending much time on designing systems (OUSTERHOUT, 1998; LOUI, 2008; DUBOIS, 1999; LANGTANGEN, 2006). In this chapter, we attempt to understand why and how some scientists use scripts in experiments by analyzing the state-of-the-practice.

This chapter is organized as follows: Section 2.2 runs a questionnaire with scientists for understanding what their most preferred tool is and what are their reasons for the preference. Section 2.3 analyzes real experiment scripts from the scientific repository DataOne to understand how scientists use scripts. Finally, Section 2.4 discusses the results and presents the final remarks.

### 2.2 Questionnaire

While many tools exist to support scientists, we have little insight on which scientists frequently adopt and how do they use them. This information could help in the development of new tools and processes, focusing on who would use them. Previous work has tried to shed some light on this subject before. Hannay et al. (2009) ran a questionnaire with nearly 2,000 scientists to understand how do they develop and use scientific software. While they present relevant findings regarding the importance of scientific software, the usage of software engineering practices, the size of scientific software, and the lack of formal training, they do not analyze which tools scientists use and why do they use these tools. Pinto, Wiese, and Dias (2018) recently replicated

this questionnaire with more than 1,500 responses from a population of R developers. Most results were consistent with the original study. In addition to R, the respondents also indicated that they use C/C++, Python, or Shell Script to develop scientific software. Finally, Prabhu et al. (2011) ran a questionnaire with 114 researchers and identified a high usage of MATLAB and Fortran, with scientists combining both and other programming languages to achieve better performance. They also identified that scientists run experiments that take days in desktops and that they need tools to analyze and enhance the performance of experiments.

These previous work indicate that scientists use and combine different computational tools to achieve their goals, but do not analyze why do they prefer these tools and which features do they use. Thus, the goal of this section is to answer the following research questions:

- RQ.Q1 – What are the scientists’ preferred/more often used tools to run experiments?
- RQ.Q2 – Which tool is their favorite?
- RQ.Q3 – What are the reasons for the preference?

We ran a questionnaire to answer these questions. We obtained 120 answers in the questionnaire, and from these answers, we discovered that these scientists prefer to use scripting languages due to their easiness to set up and run, flexible development, and previous experience of the scientists. More specifically, the most used language is Python.

In the following subsections, we present the material and methods used for the analyses, followed by their results. We also discuss threats to the validity of this work.

## 2.2.1 Materials and Methods

In this section, we discuss the methodology we used to design a questionnaire for understanding which computational tools scientists use to run experiments and what is the reason for their preference.

### 2.2.1.1 Distribution

For answering the research questions, we designed a questionnaire and made it available online with a custom implementation that allowed us to obtain partial answers and use answers from people who dropped out in the middle of the questionnaire (PIMENTEL, 2017).

In the questionnaire design, we aimed at reducing at maximum the dropouts by reducing the participant effort. Thus, our questions presented prefilled choices and included an extra option

to write other answers. Some of the questions allowed participants to select multiple answers, while others were restricted to only one. Additionally, the participants could skip any question that they did not feel comfortable answering.

We distributed the questionnaire to the following scientific companies and communities: Fiocruz, LNCC, INRIA-Montpellier, SciPy, DataOne, Software Carpentry, e-Science Oxford, and DB-BRAS. We also distributed to Universities: UFF, UFRJ, USP, NYU, UPenn, University of Southampton, Newcastle University, and University of Amsterdam. Finally, we asked everyone to share it with other colleagues. In total, 120 respondents answered at least one question of the questionnaire from March 13, 2017, until December 13, 2017, but only 64 people answered all questions. It occurred not only because some people dropped out of the questionnaire, but also because some answers skipped other questions. Figure 2.1 presents the number of participants that answered each question. Note that we divided the questionnaire into two parts: the “P” part characterizes the participants, and the “R” part answers the research questions.

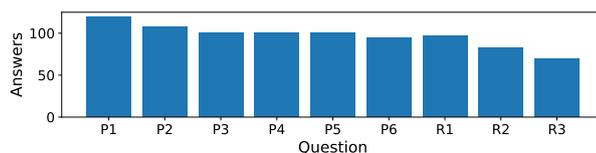


Figure 2.1: Number of participants that answered each question.

### 2.2.1.2 Population

In the first part of the questionnaire, we had the goal of characterizing the participants and use this information to characterize the scientists who perform computational experiments. With this goal in mind, we elaborated the following questions:

- P1 — What is your education level?
- P2 — How many scientific experiments have you ever performed on computational environments?
- P3 — What are your scientific domains?
- P4 — How much experience do you have in running scientific experiments on computational environments?
- P5 — In which roles have you performed computational experiments?
- P6 — What is your country of residence?

Initially, we had 120 participants that answered at least one question. However, we removed 19 participants that indicated that they have never performed computational experiments. We detected these participants by checking whether they answered zero or skipped P2 (How many scientific experiments have you ever performed on computational environments?). We also preemptively stopped the questionnaire for all people who answered zero in P2. In addition to these removals, we removed one participant that asked for a clarification of what we meant by “experiment” in the questionnaire. Hence, we consider only the answers of 100 participants.

Figure 2.2 presents the answers to P1 (What is your education level?) and P2 (How many scientific experiments have you ever performed on computational environments?). Note in Figure 2.2a that most participants either have a Ph.D. degree or are pursuing one. Figure 2.2b indicates that most participants have performed more than ten computational experiments.

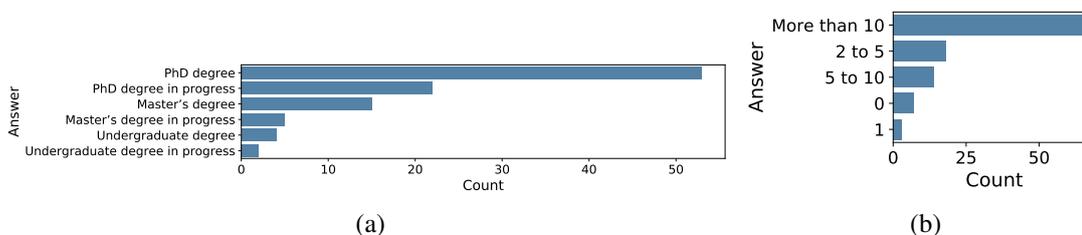


Figure 2.2: (a) P1 - Education level (100 participants); (b) P2 - Number of experiments performed in computational environments (100 participants).

Figure 2.3 presents the results of P3 (What are your scientific domains?). As expected, most participants that perform computational experiments are in the computer and information science domain. However, this figure indicates that this domain is not the only one that performs computational experiments. It also shows that a significant number of biological sciences researchers, engineers, and geoscientists run computational experiments.

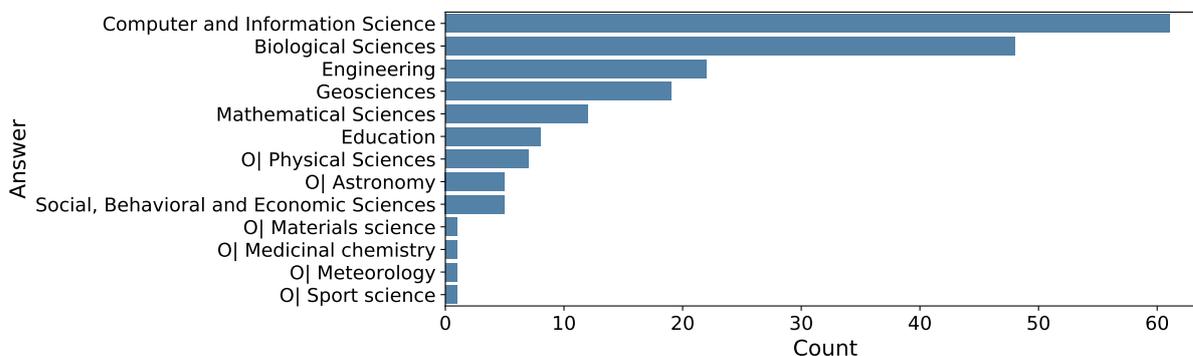


Figure 2.3: P3 - Scientific domains (100 participants). Answers starting with “O|” were not predefined in the questionnaire.

Figure 2.4 presents the results of P4 (How much experience do you have in running scientific experiments on computational environments?). Most participants have more than two years of experience. It was not surprising, since most participants have a Ph.D. degree, according to P1, and Ph.D. degrees usually take more than three years to pursue. In fact, by analyzing P5 (In which roles have you performed computational experiments?) in Figure 2.5, we can see that many participants performed computational experiments during their Ph.D. and Master courses. Figure 2.5 indicates that we received more answers from academic roles than company roles. Like P3, P5 accepted multiple answers. Finally, Figure 2.6 presents P6 results (What is your country of residence?). We received answers from 12 countries. The participants mainly reside in Brazil or the United States of America.

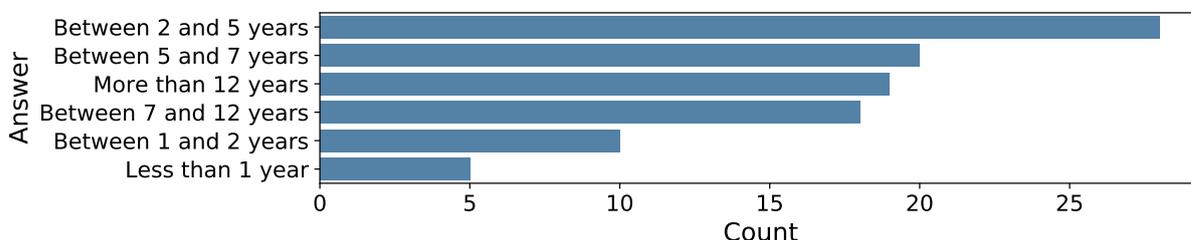


Figure 2.4: P4 - Expertise in years (99 participants).

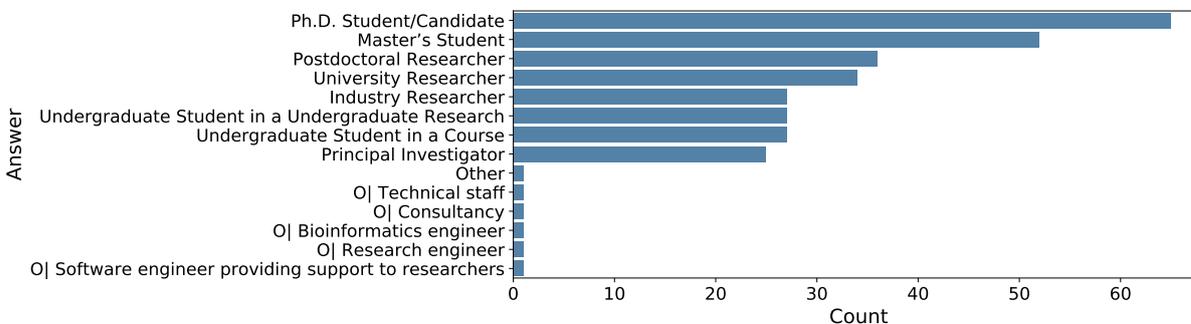


Figure 2.5: P5 - Role of participants when they performed computational experiments (98 participants). Answers starting with “O|” were not predefined in the questionnaire.

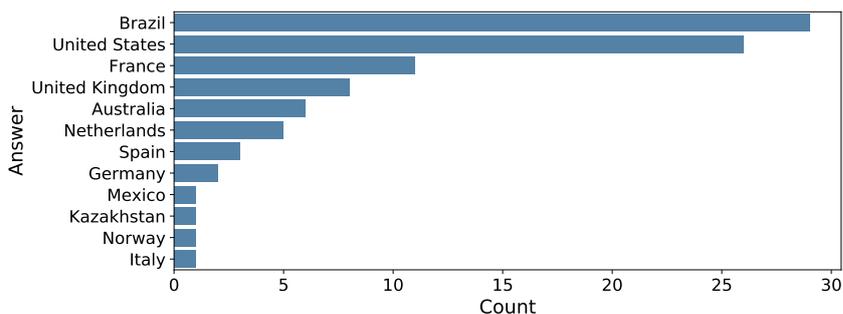


Figure 2.6: P6 - Country of residence (89 participants).

## 2.2.2 Results

In the second part of the questionnaire, we had the goals of understanding which tools scientists use and prefer to use and why, to answer the research questions RQ.Q1, RQ.Q2, and RQ.Q3.

### 2.2.2.1 RQ.Q1. What are the scientists' preferred/more often used tools to run experiments?

We asked participants to select up to 3 tools in a list of 25 tools or write different ones to answer RQ.Q1 (What are the scientists' preferred/more often used tools to run experiments?). We divided these 25 tools into three categories: SWfMS<sup>1</sup>, scripting languages (script)<sup>2</sup>, and system programming languages (prog)<sup>3</sup>.

We received 95 answers to RQ.Q1 with 36 tool indications<sup>4</sup>. Figure 2.7a presents these tools grouped according to the three categories (SWfMS, Script, and Prog). The category *other* has tools that do not fit into those categories<sup>5</sup>. Note that scripting languages are the most often used tools. Figure 2.7b presents a word cloud of the answers. *Python*, *shell script*, and *R* appear to be the preferred tools. These tools are all scripting languages.

While RQ.Q1 answers the most often used tools, it is possible to see in Figure 2.7a that there are more answers to scripting languages than participants. It indicates that the participants use more than one scripting tool in their experiments. Finally, Figure 2.7c indicates that most participants combine different types of tools in their experiments. All participants that use system programming languages also use scripting languages. One participant uses only SWfMS. Thirty participants use only scripting languages. It may indicate an advantage of using scripts instead of other types of tools for a variety of experiments. Nonetheless, 67% of the participants that use scripts also use SWfMS or system programming languages. Thus, computational experiments can benefit from a combination of multiple tools.

### 2.2.2.2 RQ.Q2. Which tool is their favorite?

In the questionnaire, we presented the participant's RQ.Q1 answers as options for RQ.Q2 (Which tool is their favorite?). We skipped RQ.Q2 and RQ.Q3 should the participant select

---

<sup>1</sup>Askalon, Chiron, e-Science Central, Galaxy, Kepler, Pegasus, SciCumulus, Swift/T, Taverna, and VisTrails.

<sup>2</sup>IDL, Javascript, Julia, Matlab, Perl, Python, R, S, Shell Script, and Wolfram Language.

<sup>3</sup>C, C++, Fortran, Java, and Object Pascal.

<sup>4</sup>Other SWfMS: Tavaxy, CWL, Snakemake, Dispel4py, HTCCondor, WS-PGRADE, Wings, Girder, Girder-worker, and Luigi.

Other script language: GNU Octave.

<sup>5</sup>make, miniconda, and orgmode.

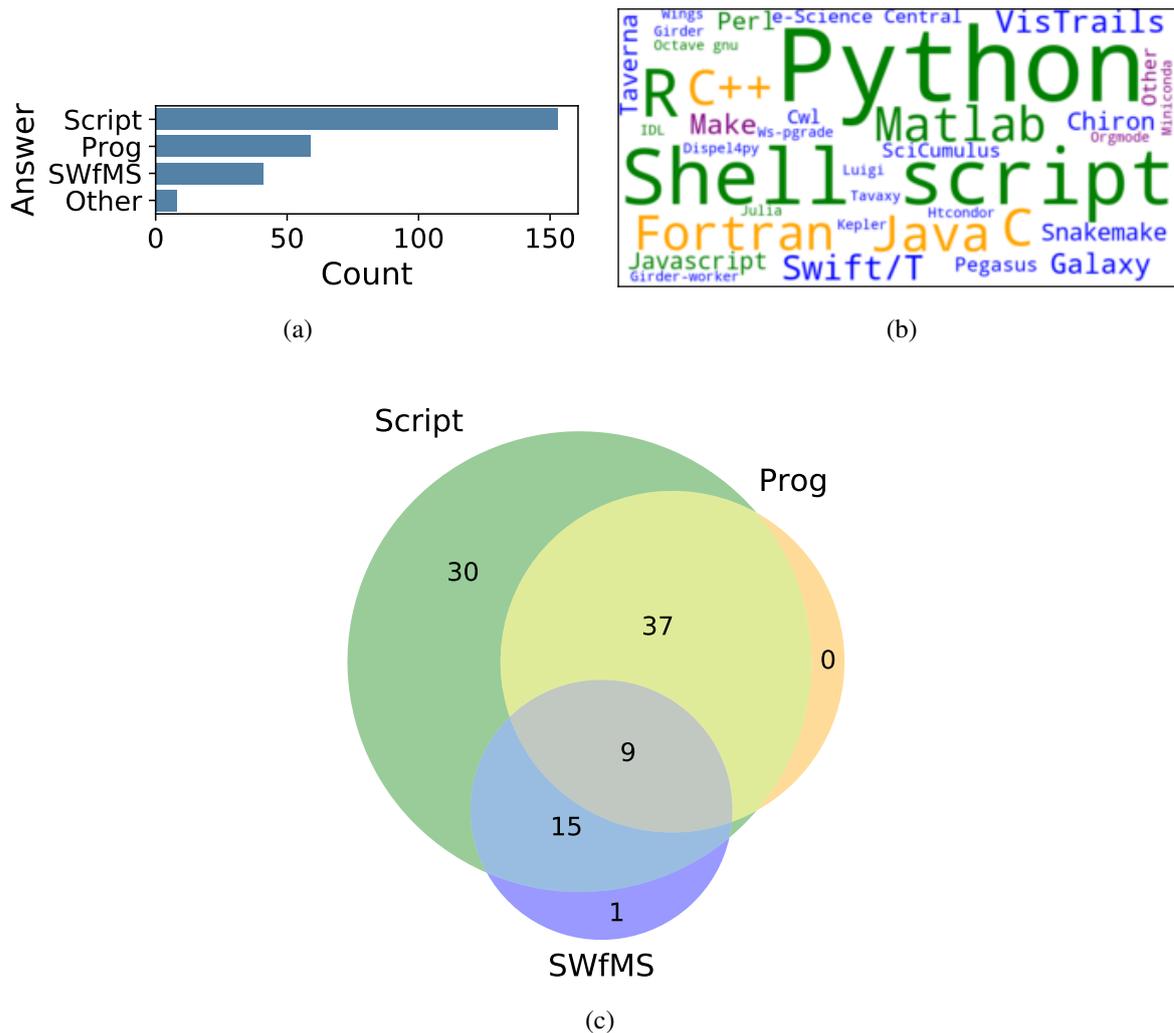


Figure 2.7: RQ.Q1 – (a) preferred/more often used tools (95 participants) grouped by categories, (b) word cloud, and (c) Venn Diagram.

a single tool in RQ.Q1. However, in this case, we used the RQ.Q1 answer as an answer for RQ.Q2. Figure 2.8 presents the RQ.Q2 results. Once again, Python is the favorite tool of the participants. By selecting participants that use Python as their main tool, we found that 94% of them also use other tools. This percentage is more significant than the one we found for scripting languages in Figure 2.7c. Hence, participants that prefer Python seem to use multiple tools in their experiments.

### 2.2.2.3 RQ.Q3. What are the reasons for your preference?

Figure 2.9 presents the answers to RQ.Q3 (What are the reasons for your preference?) normalized by the number of participants that choose each tool. We had 68 respondents to this question: 9 of them chose a SWfMS as their favorite tool; 42 chose scripting languages; 14

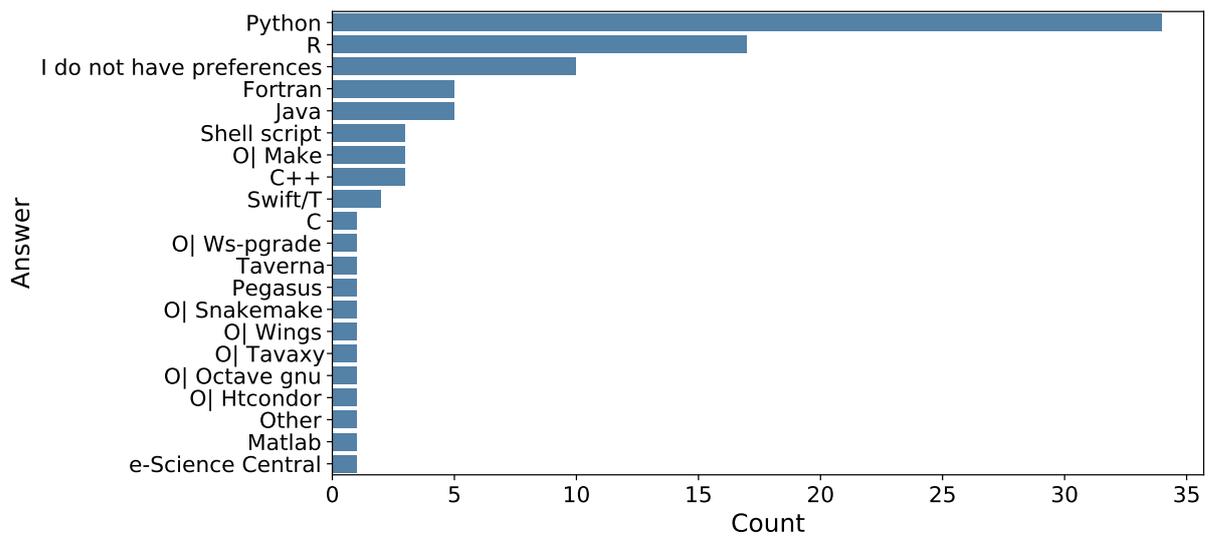


Figure 2.8: RQ.Q2 - favorite tool (92 participants). Answers starting with “O|” were not predefined in the questionnaire.

chose system programming languages; and the remaining 3 chose other tools, such as make.

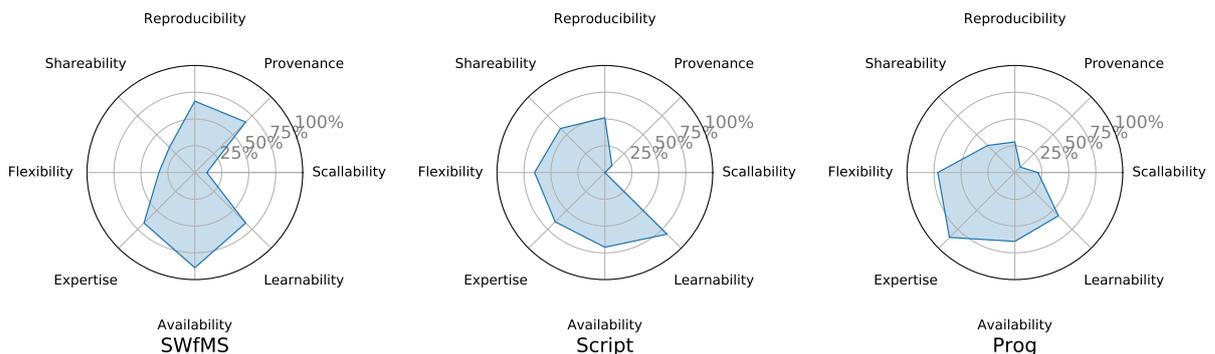


Figure 2.9: RQ.Q3 - reasons for tool preference (68 participants).

In the questionnaire, we provided a set of predefined descriptive answers and allowed the users to write their own. However, in Figure 2.9, we grouped the answers into eight categories:

- Reproducibility – easiness to reproduce the execution.
- Provenance – the ability to collect and analyze provenance.
- Learnability – easiness to learn and the existence of documentation and supporting facilities.
- Availability – easiness to set up and run in different systems and the availability of domain-specific libraries.
- Expertise – previous experience of the respondent.

- Flexibility – possibility of modifying experiments in a flexible way.
- Shareability – easiness to share.
- Scalability – the ability to use with vast amounts of data and processing. In the predefined answers, we did not write any scalability option. Thus, the participants wrote the answers that appear for SWfMS and System Programming Languages in Figure 2.9.

The four main reasons for choosing scripting languages were learnability, availability, flexibility, and expertise. System programming language had similar traits, with an indication of more flexibility and less learnability. However, while scripting languages also presented a high number of answers related to shareability and reproducibility, system programming languages fail in these aspects but succeed in scalability. While scripts are arguably worse for scalability, they may be integrated with system programming languages to improve their scalability. This happens for libraries that demand high performance, such as NumPy in Python. NumPy implements its core functionalities in Fortran for performance.

In comparison to SWfMS, scripting languages are more flexible, shareable, and learnable but less reproducible, scalable, available, and with less provenance support. SWfMS are more available than scripts because they are easy to set up and run, and are cloud friendly. The built-in cloud support also makes them more scalable than general scripting languages. Provenance and reproducibility features are more associated with SWfMS, as these systems have been offering this kind of support for a decade. On the other hand, scripts are more flexible than SWfMS. Since scripts exist for a longer time and have a user base beyond the scientific community, they have more learning material and received more investment in libraries, debuggers, and IDEs, and other features (BARKER; VAN HEMERT, 2007). Moreover, scripts do not impose lock-in restrictions, such as custom file formats that prevent you from moving to SWfMS. Hence, scripts are more shareable.

### 2.2.3 Threats to Validity

Our results have some threats to validity, which we discuss next.

**Internal.** The main threat to internal validity is the possibility of selection bias. In our effort to have the maximum number of answers, we shared the questionnaire with groups that are close to the authors. It may have biased the results towards the tools that we use. We asked people to forward the questionnaire to other groups in their institutions to mitigate this threat.

**Construct.** Confounding constructs is the main threat to construct validity. At the end of the questionnaire, we included an optional question for comments from the participants. One of the participants asked for clarifications on what do we mean by “experiments”. We removed this participant from the analyses. In addition to the optional question for comments, we made our emails available at the beginning and at the footer of the questionnaire for clarifications, but we did not receive any inquiries. In addition to confounding constructs, there is no way to guarantee that participants give the correct answers to questionnaires, which affects the reliability of measures. Finally, another threat to construct validity is the lack of scalability options in RQ.Q3. Even though we allowed respondents to write answers to this question, their choices may be biased towards the predefined options.

**External.** Representativeness of subjects is the main threat to external validity. Even though we shared the questionnaire with different groups, we did not attempt to share evenly to all the scientific communities. It is evident by the higher number of respondents from Brazil.

## 2.3 Script Analysis

As presented in Figure 2.7c, scripts are pervasive in the automation of scientific experiments, being used even by scientists that adopt system programming languages or workflow management systems. However, the actual use of scripts is largely unknown by the developers of tools and processes to support the automation of such experiments. More specifically, many participants of the questionnaire use and have Python as their favorite tool for computational experiments. While the previous section explains why they use Python, in this section, we seek to understand how they use it to improve our answer to RQ.Q3 and answer the following research questions:

- RQ.S1 – How do scientists structure scripts?
- RQ.S2 – How do scientists use modules and external tools?
- RQ.S3 – How do scientists process data in scripts?

Python is a general-purpose scripting language that has many distinct applications, such as web development (PIMENTEL, 2017), game development (MCGUGAN, 2007), and scientific experiments (SHEN et al., 2014; MCKINNEY, 2011). We obtained Python scripts from DataOne (MICHENER et al., 2011), intending to limit our analysis to scientific Python scripts, due to its relation to science.

DataOne is a cyberinfrastructure that provides open access to Earth observational data (MICHENER et al., 2011). Scientists from many domains use DataOne. In our analysis, we identified scripts that relate paralogs to the proportion of heterozygous individuals, analyze the social activities of dolphins, study how people create vocal communication systems, report the location of specific DOIs within the full text of papers, among many others. We performed a static analysis over these scripts to understand which language constructs do scientists use and which libraries do they choose for their experiments. Our analysis suggests that scientists do not use many complex programming language constructs. Instead, they use simple constructs such as variable assignments, imports, loops, and conditional control flows, in addition to built-in data-structures. Built-in modules play an important role in Python, but scientists also use other modules such as NumPy, Bio, Pandas, and Matplotlib. These modules take advantage of Python integration with C and Fortran to obtain a better performance.

## 2.3.1 Materials and Methods

In this section, we discuss the methodology we used to collect, prepare, and analyze scripts from DataOne to understand how scientists use Python for scientific experiments.

### 2.3.1.1 Analyses

For analyzing how scientists use Python for scientific experiments, we define the following research questions:

**RQ.S1.** *How do scientists structure scripts?* Python supports writing scripts with many constructs and does not enforce the usage of them all. While a script might use only built-in functions in a procedural way, another script might define functions and classes, use them as first-class objects, and use functional and object-oriented paradigms. Moreover, some constructs can appear at different positions in the script and follow different styling guides. For instance, the position of *imports* and *function definitions* can be either at the top of the script or mixed with other statements. Similarly, users can write the main execution code within the main function; write directly on the body; or write directly on the body within a guard that prevents it from being executed when it is only meant to be imported by other scripts. Finally, users can use comments on scripts to provide metadata for other programs (e.g., a shebang on top of the script specifies which interpreter should run it), to disable the execution of some statements, or to describe their experiments.

In this analysis, we use both the built-in `ast` module and the `astroid` (AUTHORITY,

2017) module to identify which constructs scientists use in scripts. Additionally, we carefully inspect the scripts to identify where and how scientists define *imports*, *functions*, *classes*, and *comments*, and whether they use functions as first-class objects. Finally, we check how scientists use comments in scripts.

**RQ.S2.** *How do scientists use modules and external tools?* Python has “batteries included” (DUBOIS, 2007). It provides many functions and modules for users to use without any external module or tool. However, these functions and modules are often not complete enough for scientific experiments, and scientists need to import external modules or use external tools. Additionally, scientists can also define their own modules with helper functions. In this analysis, we check which modules scientists use, identifying whether they are built-in, user-defined, or external. We also identify whether or not scientists use external tools.

**Mining Relationships between modules.** We analyze which modules are used together by mining association rules (AGRAWAL; SRIKANT, 1994). Association rules have the goal of finding probabilistic associations or correlations. They are expressed as  $X \rightarrow Y$ , where  $X$  is the antecedent set, and  $Y$  is the consequence set. Their interpretation is based on the amount of evidence determined by three metrics: *support*, *confidence*, and *lift* (AGRAWAL; SRIKANT, 1994; HAN; PEI; KAMBER, 2011). These metrics can be calculated as follows:

$$\text{support}(X \rightarrow Y) = P(X \cup Y) \quad (2.1)$$

$$\text{confidence}(X \rightarrow Y) = P(Y|X) = \frac{P(X \cap Y)}{P(X)} \quad (2.2)$$

$$\text{lift}(X \rightarrow Y) = \frac{P(X \cap Y)}{P(X) \times P(Y)} \quad (2.3)$$

The *lift* metric indicates how much the occurrence of  $X$  increases the probability of  $Y$  occurring. When  $\text{lift} > 1$ ,  $X$  increases the probability of  $Y$ ; when  $\text{lift} = 1$ ,  $X$  does not interfere with  $Y$ ; and when  $\text{lift} < 1$ ,  $X$  decreases the occurrence of  $Y$  (HAN; PEI; KAMBER, 2011).

**RQ.S3.** *How do scientists process data in scripts?* There are many ways to load input data for processing in a script. Scientists can use variables or arguments to define the data, read data from the standard input, or read data from files. The path of these files can also be defined

through variables or arguments or be passed directly to `open` functions as literals. All these methods of loading the input data can be combined into a single script. Similarly, there are multiple ways to save or present the results of experiments. Scientists can write into files using the Python `open` function or an external library function, use external tools, or just print or plot the results. In addition to these aspects, scripts might produce single or multiple results. They might apply the same operations individually to a set of input files, producing distinct results for each input file, or they might combine input files into a single result. Finally, the moment of processing also varies across scripts. Some scripts process data while they read the input data, others process data while they write the output data. Some scripts read all the input data, process it, and write the output data. Finally, some scripts read, process, and write the output at the same time (i.e., at the loop that reads the input data). In this analysis, we identify all these characteristics of scripts.

### 2.3.1.2 Data Collection

Using the DataOne RESTful API, we could query all files with mimetype `application/x-python`. This query returned 199 results on December 4<sup>th</sup>, 2017, but 3 of them had the wrong mimetype and were not Python scripts. Among these scripts, we found 17 duplicated scripts and removed them from our analyses. Finally, we found that 7 scripts were user-defined libraries through manual inspection of the scripts. Since our goal is to understand scripts that define scientific experiments, we also removed the libraries. Hence, we analyze 172 valid scripts in this section.

The scripts belong to datasets with authorship data. Half of these datasets have at least 4 authors. Some authors worked on many datasets and collaborated with others. By clustering the authors that worked together, we found 67 groups. Most groups submitted at most one script. However, some groups submitted up to 15 scripts. In total, 32 groups submitted 137 scripts to DataOne.

Some scripts in DataOne have syntax issues: 36 (20.93%) scripts mix tabs and spaces, which is problematic in Python since it uses the indentation to define the scope; 5 scripts have comments that do not use the Python syntax for comments; and 5 scripts have wrong indentations. Before our analysis, we manually fixed these issues.

For analyzing the scripts, we used Python modules that require to match their Python version with the Python version of the scripts (i.e., we could not run the `ast` module in Python 2 with a Python 3 script). However, only 11 scripts declare the Python version using shebang expressions or comments. Among these, seven scripts indicate Python 3, one indicates both Python 2 and 3, and the other 3 scripts indicate Python 2. Through an analysis of language con-

structs, we found that 87 scripts have language constructs that are only valid for Python 2, such as `print` and `exec` statements without parenthesis, parameter unpacking in function definitions, except statements with a comma, and “<>” as inequality operator. Additionally, a single script is only syntactically valid for Python 3, due to the presence of assignment unpacking. This script properly declared Python 3 in the shebang. The syntax of the 84 remaining scripts is valid for both Python versions. In the analyses of this section, we only used Python 3 for scripts that declared only Python 3 as their version. We used Python 2 for all the other scripts that had a syntax valid for both versions or for Python 2 only. Note that it does not guarantee that all of these scripts run in Python 2, as some semantics also changed between these versions. However, since we only performed static analyses, choosing one over the other does not change the results.

## 2.3.2 Results

In this section, we present the results we collected to answer each question.

### 2.3.2.1 RQ.S1. How do scientists structure scripts?

Python supports many constructs, and the analyzed scripts do not use all of them. Using both the built-in `ast` and the `astroid` (AUTHORITY, 2017) modules, we obtained the syntactic constructs used in each script, and we produced the histogram of Figure 2.10. This figure groups constructs into categories. Thus, when we talk about the definitions category, we refer not only to function definitions, but also to generators, lambda (anonymous functions), decorators, and class definitions.

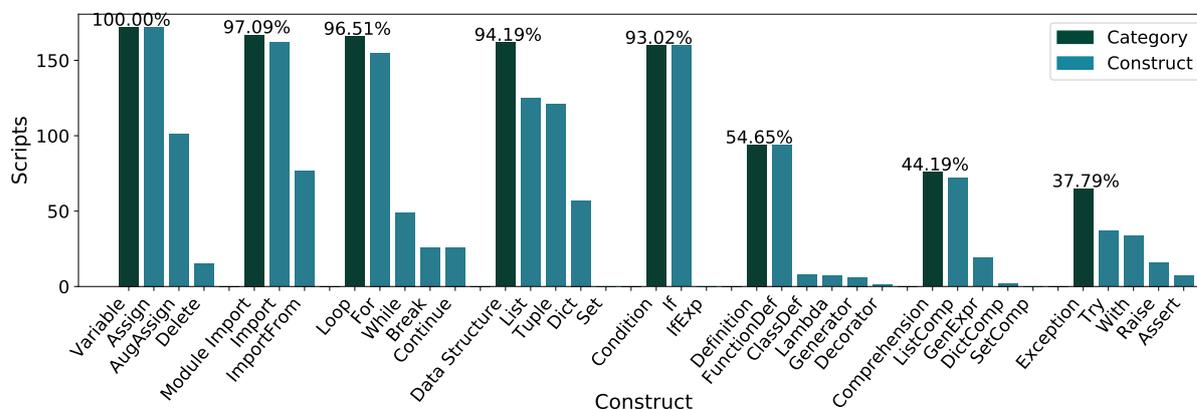


Figure 2.10: Distribution of Python constructs in scripts. This figure groups constructs into categories. The constructs of a category appear on the right of the category bar. A category corresponds to the union of its constructs.

Simple constructs, such as *variables assignments*, *imports*, *built-in data structures*, *loops*, and *if conditions* appear in almost all scripts. *For loops* represent most loops, but some scripts use *while loops* and do not use *for loops*. Most scripts use *built-in* data structures. *Lists* and *tuples* are more common than *dictionaries*, and we could not identify any script that uses the syntax for defining *sets*. However, the syntax for defining sets requires the users to create the set with at least one element, and it was added to Python syntax more recently. Nonetheless, we found 9 (5.23%) scripts that invoke the `set` function to create an empty set.

Some constructs, such as *augmented assignments*, *function definitions*, *list comprehensions*, and *exception handling* appear in about half of the scripts. The lack of function definitions in the other half of scripts may indicate that the scripts are simple enough, some scientists do not care much about abstractions and code reuse, or that functions from external modules are sufficient for data analyses.

Finally, complex constructs such as *anonymous functions* (`lambda`), *generator definitions*, *decorators*, *classes*, *dictionary comprehensions*, and *if expressions* barely appear in scientific scripts. Among complex constructs, we found 50 scripts (29.07%) that use callable objects (e.g., functions, classes) as first-class objects. Most of them (46 scripts) pass the callable objects as arguments to other functions, but we also found 10 scripts that invoke callable objects directly. In general, functions that receive the callable objects are either built-in functions or functions defined by external modules. We only found one script that defined a function that expects a callable as parameter. As for the type of the callables, we found 33 scripts that use existing type objects (e.g., `int`, `float`, `str`) and pass them as arguments to other functions; 11 scripts that obtain the callable from an operation, such as an attribute access to a method or a function call that returns a function object; 7 scripts that define their own callable objects using `lambda`; and 9 scripts that define functions and pass them as parameters to other functions. In this latter case, 3.49% scripts define parallel functions and pass them to functions that manage the parallelism.

Only knowing which constructs scientists use is not enough to answer how they structure the scripts. It is also important to know where they use these constructs. Through careful inspection of scripts, we identified four consecutive conceptual regions in the structure of scientific scripts: header, top, definitions, and bottom. Figure 2.11 presents these regions. The *header* is the region at the beginning of the scripts where they have imports, comments describing the experiment, and comments with metadata. This region usually starts with comments with metadata for the operating system or the interpreter. For the operating system, 89 scripts include shebang expressions that indicate the path of the Python interpreter that should run the script (line 1 of Figure 2.11). For the interpreter, 20 scripts indicate the encoding of the file (line 2 of

Figure 2.11). After the comments with metadata, most scripts include a description of the script in the header region. For the description, 70 scripts use the comment syntax of Python (line 3 of Figure 2.11), and 42 use *docstrings* (see line 10 of Figure 2.11). In Python, *docstrings* are the recommended way of documenting blocks of code, as the parser is aware of them while it ignores comments. Finally, after the comments, 143 scripts have imports in this region (line 5 of Figure 2.11). Most scripts use only this region for imports, but we found 7 scripts that also have imports in other regions and 24 scripts that only have imports in other regions (i.e., they do not use the header region for imports).

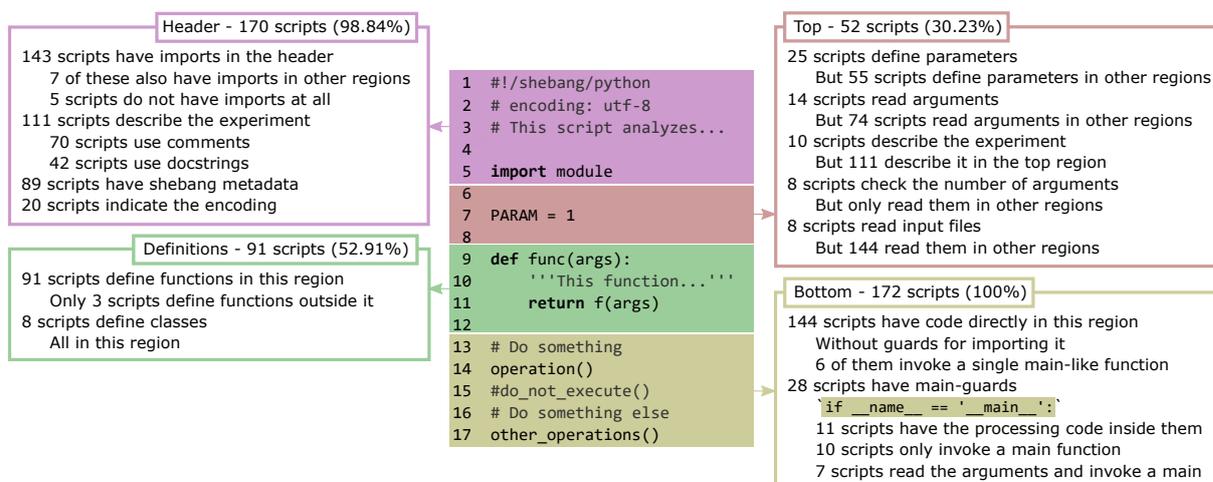


Figure 2.11: Regions of scripts. Numbers in region descriptions refer to the number of scripts that have the feature in the specified region unless stated otherwise. Note that a script may have multiple features in the same region.

The second region is the *top* region. We define this region as the region that starts after the last `import` statement, ends at the first function or class definition, and does not contain any data processing code. This region appears in 52 (30.23%) scripts, but each one of them has a different usage for the region. Additionally, the different usages of this region appear more often in other regions. For instance, while 25 scripts define parameters using variables in this region (see line 7 of Figure 2.11), 55 scripts use variables to define parameters at the beginning of the bottom region or in main functions. Similarly, while 14 scripts read arguments in this region, 74 scripts read arguments in the other regions. This region is also used by few scripts to read input files (8 scripts) and describe the experiment after the last `import` statement (10 scripts). Once again, these features appear more in other regions (144 scripts and 20 scripts, respectively). Finally, 8 scripts also use this region to check the number of arguments and halt the execution at the beginning, but they only read the arguments in the other regions.

The third region is the *definitions* region. In this region, the scripts define all of their functions and classes together (see line 9 of Figure 2.11). We only found 3 scripts that had function

definitions outside this region. These scripts had function definitions mixed with the processing code of the bottom region instead. Note, however, that many scripts do not have function nor class definitions. Instead, they only use predefined modules or built-in functions. In total, 91 scripts have function definitions in this region, and 8 scripts have class definitions. All class definitions occur in this region. Also, note that we do not consider `lambda` as constructs that characterize this region since they are expected to be used dynamically in the middle of the processing code.

Finally, the *bottom* region is where the experiment processing starts to occur. In Python, there is no concept of main function, but users can use an if statement (`if __name__ == '__main__':`) as a main-guard to distinguish which code the interpreter should run when it runs from the command-line, and which code should it run when other scripts import the file as a module. We found only 28 scripts that use this if statement as a main-guard: 11 of them have the main processing code inside it, 10 of them only invokes a single main function inside the main-guard, and 7 scripts read the arguments inside the main-guard and invokes a function with them as parameters. In addition to these scripts, we found 6 scripts that do not have a main-guard, but that only invokes a single function in the bottom region that acts as a main function. Finally, the remainder 138 scripts have all the main processing code directly on the bottom region. They do not define a main function nor use main-guards.

During the inspection of the regions, we also observed how the scripts use comments and docstrings. Without counting the comments in the header region, we found that 150 scripts have comments. Among these, 127 scripts use comments (line 13 of Figure 2.11) and 29 scripts use docstrings (line 10 of Figure 2.11) to explain parts of the experiments. Additionally, we observed that 72 scripts use comments to disable the execution of Python lines. These numbers indicate that DataOne scripts use many more comments for explaining the experiments than to manipulate the code execution.

### 2.3.2.2 RQ.S2. How do scientists use modules and external tools?

We found that 167 scripts import modules. Using the built-in `ast` module of Python, we obtained the names of the modules these scripts import. We got a total of 612 imports, referring to 102 distinct sub-modules from 82 distinct top-level modules. A top-level module may have multiple sub-modules. For instance, some scripts import the following sub-modules from the `Bio` module: `Bio.SeqRecord`, `Bio.Seq`, `Bio.Align.Application`, `Bio.Align`, `Bio.Alphabet.IUPAC`. The same situation occurs with `Matplotlib` regarding its `matplotlib.pyplot` sub-module, and with `SciPy` regarding its `scipy.stats` sub-module.

While scripts import all these modules, they often do not use all of them. On average, scripts import 3.37 distinct top-level modules. However, they only use 2.89 distinct top-level modules on average.

We classified the used modules according to their definition type and domain. We considered three definition types: built-in, external, and user-defined. Built-in modules come with the Python interpreter. External modules are provided through publicly available packages. Finally, user-defined modules are local modules that users define for themselves and only share them among the experiment files. Regarding the domain, we identified nine domains for the modules, as follows:

- System – communicates with the operating system to invoke external tools, manipulates paths, and extracts other OS data, such as date and time (e.g., `sys` and `os`);
- Math – provides functions to perform math operations (e.g., `numpy` and `math`);
- Input/Output – reads input data or writes output data. In this category, we considered modules for building command-line interfaces (e.g., `argparse`), reading files (e.g., `csv`), outputting messages (e.g., `logging`), and creating graphical interfaces (e.g., `pygame`);
- Data Structure – performs operations on data structures (e.g., `pandas` and `networkx`);
- Biology – provides functions for bioinformatics analyses (e.g., `Bio` and `dadi`);
- Pattern Recognition – recognizes patterns in data (e.g., `re` and `cv2`);
- Visualization – plots data (e.g., `matplotlib` and `pylab`);
- Interpreter – extends the interpreter capabilities and provides helper functions (e.g., `__future__` and `IPython`);
- Concurrency – supports concurrency or parallelism (e.g., `concurrent` and `multiprocessing`).

Figure 2.12a presents the top 20 used top-level modules classified according to their domains and definition type. Note that the most used modules are the built-in system modules `sys` and `os`. We found that the scripts use the `os` module to create directories, manipulate paths, and invoke programs. They use the `sys` module mostly to obtain the arguments from the command-line or to halt the execution of the script. Following these modules, the most used module was `numpy`, a module that provides many efficient math operations to work on arrays or matrices.

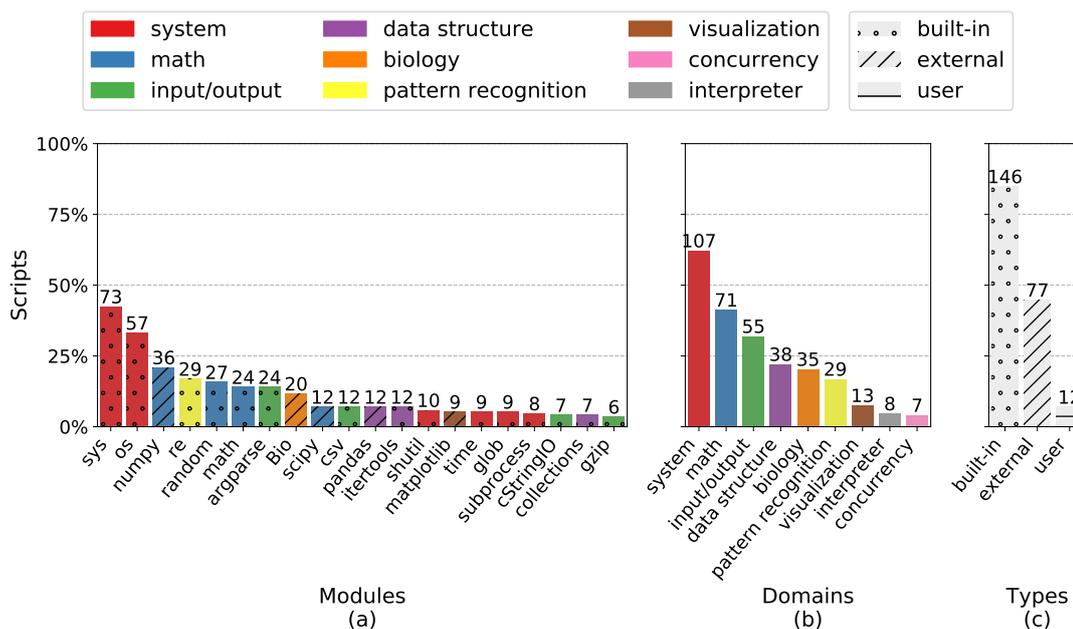


Figure 2.12: Modules – (a) Top 20 used modules, (b) All modules grouped by domains, and (c) All modules grouped by definition type.

Figure 2.12b groups all the modules into domains and presents the usage of each domain by the scripts. As expected, the system and the math domains are the most used ones. They are followed by the input/output domain, which consists of many modules used by few scripts to read and write distinct types of input and output data. Another domain with this characteristic of being composed of a high number of smaller modules is the Biology domain. The only big Biology module that appears in the top 20 of Figure 2.12a is the `Bio` module. However, many other distinct Biology modules appear in less than 5 scripts. These modules contribute to the position of the Biology domain in Figure 2.12b.

Figure 2.12c presents the usage of all modules grouped by definition type. Note that most scripts use built-in modules, which reinforces the “battery included” advantage of Python. Note also that about half of the scripts use external modules, which may indicate that only the built-in modules are not enough. User-defined modules appear in very few scripts.

Next, we identified which modules are frequently used together. Figure 2.13 presents the results classified by domain. Note that many system modules are also used together with other system modules. In fact, the most frequent set is composed of `os` and `sys`, with 28 scripts importing both these modules. However, using either of these modules is not a high indicative that the script will use the other: the lift of their association rule is only 1.16. On the other hand, other system modules that appear less frequently seem to have a strong dependency, such as the `shutil` that increases the chance of using `os` in 202% (lift 3.02), as presented in Table 2.1.

We constructed Table 2.1 by first selecting rules with only one antecedent and one consequent for simplicity. Then, sorting by lift in descending order and selecting the first 30 rules. Half of these rules represented the same rules as the other half in the opposite direction. Hence, we selected the ones with the biggest confidence and combined the others adding the column “Opposite Confidence”.

Table 2.1 also presents association rules with positive dependencies related to math modules that are used together with other math modules (e.g., `scipy` and `numpy`) and input/output modules used together with other input/output modules (e.g., `csv` and `argparse`). In the input/output domain, we found that using `StringIO` increases the chance of using `cStringIO`. It occurs because many scripts attempt to import the faster implementation (`cStringIO`) and fallback to `StringIO` when they fail. Within the other domains, we found very few modules being used together.

In the second part of Table 2.1, we found association rules between modules of different domains. We found that scripts that import `pandas` have more chance of importing `numpy`. It probably occurs because `pandas` use `numpy` underneath to perform fast math operations and allows users also to use it. Related to this reason, we found that importing `time` also increases in 219% (lift 3.19) the chance of importing `numpy`. Scientists that measure the time of their experiments may try to use faster modules to improve the speed. Finally, we found association rules related to other modules, but we could not identify a reason.

Table 2.1: Association Rules for Module Usage. Domains abbreviated as follows: *I/O* – Input/Output; *DS* – Data Structure; *Pattern* – Pattern Recognition; *Vis* – Visualization.

Antecedent	Antecedent Domain	Consequent	Consequent Domain	Support #Scripts	Confidence	Opposite Confidence	Lift
<code>StringIO</code>	I/O	<code>cStringIO</code>	I/O	6	100%	86%	24.56
<code>scipy</code>	Math	<code>numpy</code>	Math	10	83%	28%	3.98
<code>csv</code>	I/O	<code>argparse</code>	I/O	6	50%	25%	3.58
<code>math</code>	Math	<code>random</code>	Math	13	54%	48%	3.45
<code>shutil</code>	System	<code>os</code>	System	10	100%	18%	3.02
<code>glob</code>	System	<code>os</code>	System	8	89%	14%	2.68
<code>StringIO</code>	I/O	<code>math</code>	Math	6	100%	25%	7.17
<code>cStringIO</code>	I/O	<code>math</code>	Math	6	86%	25%	6.14
<code>collections</code>	DS	<code>argparse</code>	I/O	6	86%	25%	6.14
<code>itertools</code>	DS	<code>argparse</code>	System	7	58%	29%	4.18
<code>shutil</code>	System	<code>re</code>	Pattern	7	70%	24%	4.15
<code>pandas</code>	DS	<code>numpy</code>	Math	8	67%	22%	3.19
<code>time</code>	System	<code>numpy</code>	Math	6	67%	17%	3.19
<code>matplotlib</code>	Vis	<code>numpy</code>	Math	6	67%	17%	3.19
<code>Bio</code>	Biology	<code>sys</code>	System	16	80%	22%	1.88

In addition to modules, we found 27 scripts that use 35 distinct external tools. Each script that uses external tools invokes three distinct tools, on average. For invoking these tools, 15

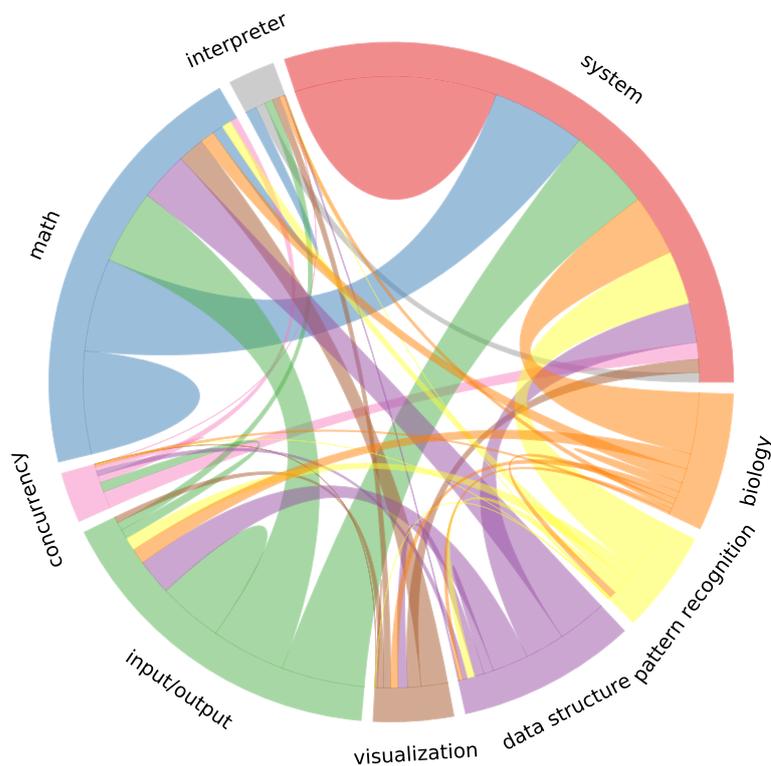


Figure 2.13: Frequent module domains used together.

scripts use the `os` module, 8 scripts use the `subprocess` module, and 5 scripts use other external modules that provide direct interfaces with the tools. We also classified the external tools according to their domains in three categories: 21 scripts use biology tools (i.e., tools that perform bioinformatics analyses, such as `Blastn`); 15 scripts use system tools (i.e., tools that come with the operating system, such as `mv` and `mkdir`); and 10 scripts invoke external interpreters (e.g., `R`, `perl`, and even `Python`).

### 2.3.2.3 RQ.S3. How do scientists process data in scripts?

In general, a scientific script loads the input data, processes it, and generates the output results. However, scripts vastly change in the way they perform these operations. For instance, while some scripts read input data from files and only process the existing data, other scripts define parameters using variables and generate new data through simulations. Additionally, the process is not always as straightforward as inputting, processing, and outputting sequentially. Some scripts combine the processing step with other steps. For this question, we analyzed these aspects of scripts.

Regarding the input step, we found that 152 scripts read data from files and 119 scripts use other types of input, such as arguments (54 scripts), variables with pre-defined data (70 scripts),

asking the user through the standard input (4 scripts), and even reading real-time camera data (0.58% script). Most scripts use more than one type of input. In fact, 99 scripts use both files and other types of input. For reading input files, scientists must specify their paths, and we identified five ways of doing so, as presented in Table 2.2. Note in this table that most scripts receive files as arguments. It indicates that some scientists develop the scripts to apply and obtain results from distinct files. Note also that the 4 scripts that read paths from the standard input are the scripts that read other types of arguments from the standard input. Similarly, 48 scripts that read paths from arguments also read other input data.

Table 2.2: Ways of defining paths for input and output files. The Input and Output columns indicate the number of scripts that apply these ways for input and output files, respectively.

Way	Input	Input (%)	Output	Output (%)	Description
Argument	81	47.09%	44	25.58%	Receive file paths as arguments.
Literal	40	23.26%	48	27.91%	Pass the file paths to functions that read the files (e.g., <code>open</code> ) directly as a string literal.
Variable	22	12.79%	12	6.98%	Pre-define variables with the paths, usually at the Top region or at the beginning of the Bottom region.
Operation	19	11.05%	65	37.79%	Perform operations to define the paths, such as combining an input argument to a string.
Stdin	4	2.33%	2	1.16%	Read the paths from the standard input.
<b>Total</b>	<b>152</b>	<b>88.37%</b>	<b>147</b>	<b>85.47%</b>	

The same ways that scientists use to define paths of input files are also used to define paths of output files in the output step. Table 2.2 also presents the number of scripts that define output paths using each one of these ways. Note in this table that 147 scripts save results to output files. For writing the output files, 115 scripts use the Python `open` function, 32 scripts use module functions, 24 scripts use external tools, and 3 scripts use the built-in `argparse` module. This module builds a command-line interface for scripts and allows users to specify the type of arguments. When users specify the argument as a file, the module already invokes the `open` function for opening it for writing. Instead of generating outputs to files, 4 scripts open windows with plots, one script uses the analyzed data to define the state of a circuit pin, and 20 scripts only print results to the standard output. Note, however, that 73.26% of the scripts use the standard output. Some use it in conjunction with files and plot more results to output. Others use it for displaying error messages, and few use it for asking for the standard input data. Finally, we found 2 scripts that did not produce output at all. These scripts had their bottom with many lines of code commented out.

Note also in Table 2.2 that more scripts use operations to define the path of output files

than they use for input files. It occurs because many scripts generate multiple output files, and defining the name of all output files through arguments would be exhausting in many cases. In total, we found 81 scripts that generate multiple results. We count multiple results as multiple files or combining a file with prints or plots. Nonetheless, not all of these scripts generate multiple results for single input or by combining data of a set of inputs. We found 39 scripts that apply the same set of operations for independent sets of inputs in a loop to generate an independent set of outputs.

Finally, we observed how the scripts process the data, and we identified four strategies: during input (22 scripts), during output (49 scripts), in the middle (71 scripts), and interweaving both input and output (54 scripts). Figure 2.14 presents examples of these strategies. With the *input* strategy, the scripts process the data while they iterate through parts of the input data (e.g., rows of table files) and store the results into a variable (`result` in Figure 2.14a). Later, they only output the already processed results (lines 6 and 7 of Figure 2.14a). With the *output* strategy, the scripts read all the data into variables without processing it (except for occasional data type conversions or data structure building). Later, they iterate through the data for processing it and outputting it at the same time. With the *middle* strategy, scripts also read all the data without processing it. Then, they process the data as its own step and store the processed data into a variable. In the end, they only output the already processed data. This strategy is hugely employed by scripts that use external modules for reading or writing data into specific formats since these modules provide single-purpose functions for the input and output steps. Finally, with the *interweaving* strategy, the scripts do not read all the data at once for processing. Instead, while they read the input data, they already process and output the results. Many scripts designed to filter data from input files use this strategy.

In addition to these strategies, we found that 18 scripts combine strategies. Some of them use the *input* strategy to read and process some data at once but also perform another processing in the *middle* or during the *output*. Others apply the *middle* strategy for a file and the *interweaving* strategy for another. Finally, some scripts apply the *interweaving* strategy for a main input

<pre> 1 for value in read(): 2     result.append( 3         process(value) 4     ) 5 6 for value in result: 7     write(value) </pre> <p style="text-align: center;">(a)</p>	<pre> 1 for value in read(): 2     data.append(value) 3 4 for value in data: 5     write( 6         process(value) 7     ) </pre> <p style="text-align: center;">(b)</p>	<pre> 1 for value in read(): 2     data.append(value) 3 4 result = process(data) 5 write( 6     result 7 ) </pre> <p style="text-align: center;">(c)</p>	<pre> 1 for value in read(): 2     result = process( 3         value 4     ) 5 write( 6     result 7 ) </pre> <p style="text-align: center;">(d)</p>
--	--	--	--

Figure 2.14: Strategies for processing data: (a) during input, (b) during output, (c) in the middle, (d) interweaving both input and output.

file. However, while they are processing and generating outputs for each row of this file, they also open other files and employ the *input* and the *middle* strategies for them.

### 2.3.3 Threats to Validity

**External.** For restricting our analyses to scientific scripts, we limited our search to DataOne scripts. Even though we obtained all scripts available at DataOne, only 172 scripts are not sufficient to understand all the specifics of scripts that represent experiments. Considering we could group authors into 67 groups, the variability of the scripts is further reduced. Additionally, most scripts belong to the biology domain, which restricts more the variability. Hence, the variety of scripts may not represent the usage of Python scripts by the scientific community in general.

**Internal.** Similarly, using only DataOne as a source for scientific scripts possible results in a selection bias. Scripts shared at DataOne may not represent the scripts that scientists write and use in the wild. Instead, they may organize it to look more shareable in a scientific repository. While obtaining scripts from other general-purpose repositories could result in a wider variability of scripts, it could also lead to the analysis of false positives (i.e., scripts that are not part of experiments nor written by scientists). Choosing DataOne reduces the number of false positives. We could also have asked scientists to provide scripts. However, this approach would lead to scripts from groups close to the authors with no guarantee that the received scripts represent the scripts scientists use in the wild. Choosing an independent service such as DataOne mitigates this threat.

Additionally, we obtained most characteristics of the scripts by reading and analyzing the scripts carefully in an iterative way. Once we noticed a pattern in some scripts, we went back to previous ones to check if the pattern existed or not. Hence, the amount of data that we observed for each script at the beginning of the experiment was different from the amount of data that we observed at the end. It could result in an internal threat to validity caused by instrumentality. We attempt to reduce this threat by re-validating the collected data.

**Construct.** Some constructs were confounding when we analyzed the scripts. For instance, when we looked at print statements to understand how scientists output results, we found print statements that were not used with this purpose. We also found other methods of printing to standard output that do not involve print statements, such as writing into `sys.stdout` or using external libraries. We attempted to reduce this threat by defining rules for what we were looking for. In the case of print statements, we separated the statements into categories and also observed other known methods that print to the standard output.

## 2.4 Discussion

In Section 2.2, we found that most people that answered our questionnaire have high education levels. Such education levels are associated with the roles in which they run experiments. While we received answers from many scientific domains that run computational experiments, the domains with more answers are computer and information sciences, and biological sciences.

In the questionnaire, we also observed that the most preferred tools of our participants for computational experiments are scripting languages. More specifically, we concluded that Python is the favorite tool among the participants of our questionnaire. It occurs due to the availability of built-in collections and modules and the flexibility for development that supports integration with other tools. Additionally, we observed that scripting languages have almost no role in provenance and reproducibility when compared to workflow management systems. It may indicate the lack of tools that aim at guaranteeing the reproducibility of experiments composed using scripting languages.

By analyzing scientific Python scripts in Section 2.3, we observed that most scientists usually adopt only simple language constructs. However, some of them deviate from that and use other complex constructs, such as functions as first-class objects. We also observed that scientists internally divide the scripts into four regions with some types of constructs appearing only in two of them: imports and comments with metadata appear all together at the beginning of the script, and functions and class definitions also appear all together in the middle of scripts. These regions help to minimize the drawbacks of flexible environments and help to organize and understand the implementation. Having a better understanding of experiments may also help in the learning process. Learning was the most selected reason for script preference.

Despite having these regions for constructs, Python scripts allow scientists to read, process, and write data using distinct strategies in a very flexible way. Moreover, the high number of distinct modules used by scripts is another indicator of the high flexibility of Python and its power in gluing solutions, which may also contribute to the preference for Python among those scientists. The capability of flexible development was the third most selected option for choosing scripts in RQ.Q3.

Additionally, many scripts use built-in modules (see Figure 2.12) and built-in collections (see Figure 2.10). All these built-in features of Python contribute to its availability, which was the second most selected option for choosing scripts in RQ.Q3.

In the script analyses, we could not find evidence of the usage of provenance in scripts.

# Chapter 3

## State-of-the-Practice: Notebooks

### 3.1 Introduction

In Chapter 2, we found that many scripts use comments to explain parts of the experiment. This characteristic of experiment scripts is related to the literate programming paradigm. This paradigm seeks to help in the communication of programs (KNUTH, 1984) by interleaving formatted natural language text, executable code snippets, and computation results. Code snippets generate the computation results, and natural language text explains both the code and the results.

However, while scripts can use comments with natural language text to support the literate programming paradigm, they are not the most appropriate tools for generating and re-using intermediate computation results. Hence, Interactive Notebooks that use scripts come into play. Interactive Notebooks are tools based on the literate programming paradigm.

Jupyter Notebook is the most widely-used system for interactive literate programming (SHEN et al., 2014). It was designed to make data analysis easier to document, share, and reproduce. The system was released in 2013, and today there are over 9 million notebooks in GitHub (PARENTE, 2020). Jupyter originated from IPython (PÉREZ; GRANGER, 2007) and, in addition to Python, it supports a variety of programming languages, such as Julia, R, JavaScript, and even system programming languages, such as C. It also allows the interleaving of not only code and text, but also different kinds of rich media, including image, video, and even interactive widgets combining HTML and JavaScript.

Kluyver et al. (2016) advocate the usage of notebooks for publishing reproducible research due to their ability to combine reporting text with the executable research code. However, the format has been increasingly criticized for encouraging bad habits that lead to unex-

pected behavior and are not conducive to reproducibility (POMOGAJKO, 2015; GRUS, 2018; MUELLER, 2018). Among the main criticisms are hidden states, unexpected execution order with fragmented code, and bad practices in naming, versioning, testing, and modularizing code. In addition, the notebook format does not encode library dependencies with pinned versions, making it difficult (and sometimes impossible) to reproduce the notebook. These criticisms reinforce prior work, which has emphasized the negative impact of the lack of best practices of Software Engineering in scientific computing software (WILSON et al., 2014), regarding separation of concerns (HÜRSCH; LOPES, 1995), tests (MYERS et al., 2004), and maintenance (HORWITZ; REPS, 1992).

Studies have been carried out to better understand how notebooks are used (KERY et al., 2018; NEGLECTOS, 2018; RULE; TABARD; HOLLAN, 2018; WANG; LI; ZELLER, 2020; KOENZEN; ERNST; STOREY, 2020). While these studies have been carried out to understand better how notebooks are used, they did not attempt to execute the notebooks and assess characteristics related to reproducibility. Hence, unlike prior work, we analyze not only the quality, but also the reproducibility of Jupyter Notebooks, and try to identify (and quantify the use of) practices that hinder reproducibility.

In this chapter, we analyze a large corpus of notebooks to obtain insights into what contributes to their reproducibility or lack thereof. We used the aforementioned criticisms as a guide to define metrics that reflect the extent of the adoption of both good and bad practices. To get more insight into the context in which good and bad practices are applied, we select a subset of popular notebooks (based on the number of stars and forks of the repositories they belong to) and compare their results with the overall results. We select this subset with the expectation that notebooks that receive more stars and forks are likely to be of higher quality.

Having the same goal of getting insight into the context of notebooks, we perform a systematic sampling of real notebooks. We perform an in-depth analysis of the notebooks in the sample, looking either for characteristics that are hard to extract automatically from the dataset or qualitative characteristics that are impracticable to manually analyze in a set of over a million notebooks. We also use notebooks from the samples to illustrate good and bad practices.

To assess the reproducibility rate, we attempt to execute the notebooks using different execution orders, isolating dependencies, and exploring strategies to compare notebook results. Finally, to gain insights into factors that influence reproducibility, we mine association rules (RUGGIERI; PEDRESCHI; TURINI, 2010) that relate specific notebook features to both success and failure of reproductions.

This chapter is organized as follows. Section 3.2 provides some background about literate

programming and Jupyter Notebooks. Section 3.3 presents the methodology of our analysis. We present the analysis results in Section 3.4. Section 3.5 presents threats to the validity of our study. Finally, Section 3.6 discusses the results.

This chapter was published in the International Conference on Mining Software Repositories (PIMENTEL et al., 2019b), and an extended version was accepted in the Empirical Software Engineering (PIMENTEL et al., 2021).

## 3.2 Background

Knuth (1984) introduced the *literate programming* paradigm that, by combining code and natural language, allows programmers to document a program’s logic. This paradigm enables the programmers themselves and others to more easily understand the code. The original system was designed for static documents and required two compilation processes (KNUTH, 1984): tangling and weaving. The tangling process executes the code snippets in the document and produces the results. Then, the weaving combines the text, code snippets, and results to deliver a human-readable document. Nowadays, literate programming is used in *interactive computational notebook environments* (SHEN et al., 2014). These environments allow parts of a notebook to be executed with immediate visualization of results and formatted text, avoiding the need for tangling and weaving.

A *Jupyter Notebook* (SHEN et al., 2014) is both an interactive literate programming document and an application that executes the document. In this work, to avoid the ambiguity, we use the term *Jupyter* to refer to the application that executes notebooks, such as *Jupyter Notebook* and *Jupyter Lab*. We use the terms *Notebook* or *Jupyter Notebook* interchangeably to refer to the literate programming document.

A notebook is composed of *cells*, which can be of three types: code, Markdown, and raw. A *code* cell contains executable code used to produce results. A *Markdown* cell contains formatted text. Finally, a *raw* cell contains text that is neither code nor formatted text – tools that convert notebooks into other formats use raw cells for configuration.

Jupyter uses a *kernel* to execute code cells. During the execution of a cell, the kernel communicates with Jupyter to display partial and final results. By default, Jupyter displays text, images (PNG, JPG, and SVG), HTML with JavaScript, and Markdown. Additionally, it supports extensions to display other formats. The notebook format uses JSON to store all of its contents in “.ipynb” files. When Jupyter sends a code cell for execution, it marks the cell as *executing* by assigning “\*” to the cell *execution counter*. After the execution, the kernel

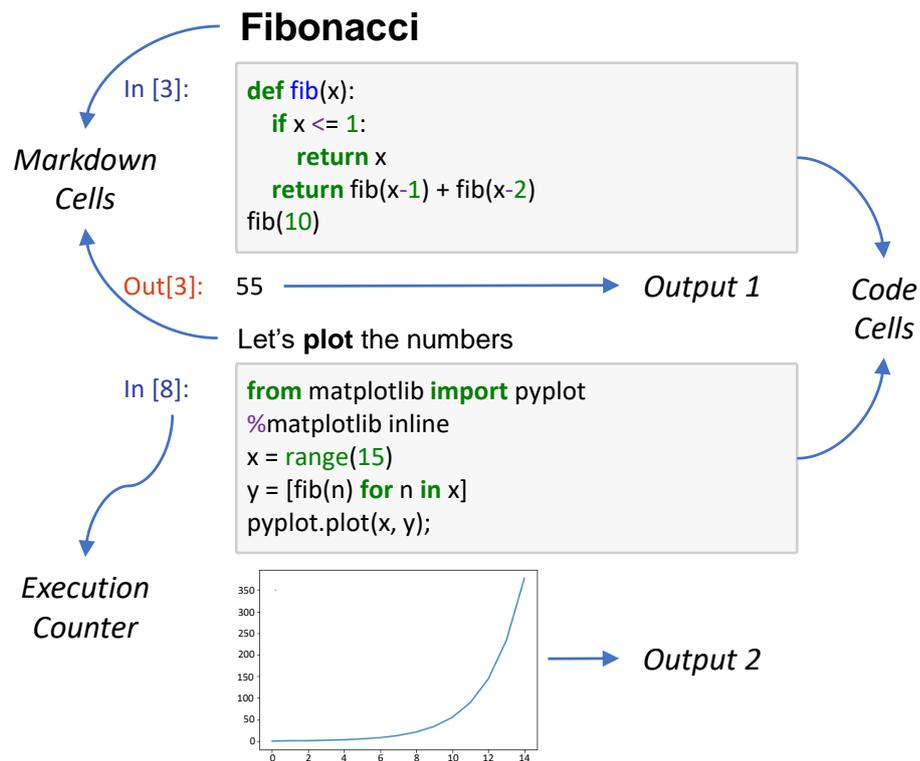


Figure 3.1: An example of an executed notebook with Markdown, code, and output.

allocates a number to the counter, which indicates the execution order. Users can execute the cells in any order, and a given cell can be executed multiple times.

Storing either *executed* or *non-executed* notebooks is possible. A non-executed notebook contains only *prospective* data (FREIRE et al., 2008), i.e., the notebook title and definition of its cells. An executed notebook contains *prospective* data plus *retrospective* data (FREIRE et al., 2008) derived by the execution of the notebook cells – the output of code cells and their execution counters. The execution of a notebook does not require cleaning the outputs of previous executions. Thus, an executed notebook may contain retrospective data from *multiple* executions.

Figure 3.1 shows an executed Jupyter notebook, which contains two Markdown cells and two code cells. On the left of code cells, Jupyter displays an execution counter that indicates the order in which the cells were executed. Below the code cells, Jupyter displays their outputs. Note that the first code cell returns a number, identified by `Out[3]`, and the second code cell displays an image without returning it. This figure also illustrates skips on the execution counters. A *skip* represents cell executions that do not have explicit definitions in the notebooks. In this case, the two executions before the execution counter 3 represent one skip, and the four executions between 3 and 8 represent another.

When initially released, IPython (PÉREZ; GRANGER, 2007) notebooks supported only Python. The system has evolved into Jupyter, which is language agnostic. Today IPython is the kernel that Jupyter uses for executing Python code. IPython supports a superset of Python. In addition to all Python constructs, it supports *line magics* to execute IPython related commands; *cell magics* to modify the semantics of code cells; *bang expressions* to execute system commands; *cell referencing* to reference the code and output of other code cells; and *help queries* to access the documentation and source code of functions and classes. Note that the second code cell of Figure 3.1 uses the line magic `%matplotlib inline` to enable the visualization of *matplotlib* figures.

### 3.3 Materials and Methods

As discussed before, Jupyter has recently been the target of substantial criticism for encouraging bad coding habits and practices that hinder reproducibility (GRUS, 2018; MUELLER, 2018; POMOGAJKO, 2015). In what follows, we discuss these criticisms and propose analyses to quantify their impact on notebooks available in GitHub (Section 3.3.1). Section 3.3.2 describes the collection and preprocessing of the GitHub data. Section 3.3.3 discusses the selection of a popular set of notebooks that we obtained to use as a baseline and compare it with the overall results. Section 3.3.4 presents the sampling process we used to gain more insights about the data. Finally, Section 3.6 discusses the corpus we use in this chapter.

#### 3.3.1 Research Questions and Analyses

The notebook criticisms relate to both prospective (i.e., code definition) and retrospective (i.e., code execution) components of notebooks (FREIRE et al., 2008). We thus frame our analyses in terms of seven research questions (RQ.N1, RQ.N2, RQ.N3, RQ.N4, RQ.N5, RQ.N6, and RQ.N7), which we organize into two categories: Analysis of Prospective Data, which covers RQ.N1-RQ.N4, and Analysis of Retrospective Data, which covers the remaining questions.

**Analysis of Prospective Data** Notebooks store cell definitions and the notebook title as prospective data. In our analyses, we used this information to answer the following questions:

**RQ.N1. *How do notebooks use literate programming features?*** According to Wilson et al. (2014), scientists should write programs for people and not for computers. Being a literate programming tool, Jupyter can fulfill this goal. Jupyter allows users to write Markdown cells with

text describing the logic behind their programs, followed by direct visualizations of the results. However, the ability to do it does not imply that users will write descriptions or whether these descriptions are meaningful. Grus (2018) pointed out that among the officially recommended tutorials written in Jupyter, there are tutorials with descriptive text that does not correctly explain what the code does. We analyze whether Jupyter is used as a literate programming tool by looking at the number of Markdown cells and their positions in the notebooks. Investigating the presence of linguistic anti-patterns (ARNAOUDOVA; DI PENTA; ANTONIOL, 2016) or whether the Markdown descriptions are meaningful for the notebooks is outside the scope of this work.

**RQ.N2. *How are notebooks named?*** By default, Jupyter creates notebooks titled “Untitled”. It discourages users from choosing meaningful names (GRUS, 2018). Also, the title is used as the name of the file which stores the contents of the notebook. Using the title as a filename creates OS-based restrictions in the size of titles and the allowed characters (e.g., in Windows, it is impossible to create or use a notebook that has “?” in the title (MICROSOFT, 2018)). Moreover, the choice of notebook title is restricted by the filename conventions adopted by different OS (e.g., not using space characters (TIM; DOORKNOB, 2014)). We analyze the number of untitled notebooks, the number of notebooks with “-Copy” in the title, the size of notebook titles, and the presence of characters not recommended by the POSIX fully portable filenames guide (the guide recommends A-Z a-z 0-9 . \_ -) (LEWINE, 1991).

**RQ.N3. *How do notebooks use modules, functions, and classes?*** In traditional programming languages, modules, functions, and classes are essential constructs to maintain the separation of concerns in software (HÜRSCH; LOPES, 1995). In literate programming environments, Markdown cells could be used to separate the concerns. However, this would lead from the lack of referencing and reusability. Moreover, Python treats every script as a module and allows users to import functions and classes from them, which improves the reusability across scripts. However, importing notebooks is hard and unusual (GRUS, 2018). We extract the Python Abstract Syntax Tree (AST) from cells to analyze the presence of local module imports, and function and class definitions as evidence of separation of concerns.

**RQ.N4. *How are notebooks tested?*** Testing is a good practice to verify that a given program meets its requirements and keeps working after changes are applied (MYERS et al., 2004). Since notebooks are not modules, testing code in a notebook is challenging as it requires mixing test code with the notebook narrative code (GRUS, 2018; MUELLER, 2018). To search

for evidence of testing in notebooks, we analyze the imported modules names that contain “test”, “Test”, “TEST”, “mock”, “Mock”, or “MOCK” as a sub-string. We also checked for known Python testing tools that do not have these sub-strings (i.e., antiparser, aspectlib, behave, doublex, fit, fudge, fusil, hypothesis, lettuce, ludibrio, mox, nose, peckcheck, pester, pry, pythoscope, reahl.tofu, reahl.stubble, sancho, subunit, taof, twisted.trial). We obtained this list of modules from the categories unit testing tools, mock testing tools, fuzz testing tools, and acceptance testing tools of the Python testing tools taxonomy page (PYTHON-WIKI, 2019).

### **Analysis of Retrospective Data**

Notebooks store cell outputs and execution counters as retrospective data. We use the following questions to explore the retrospective data.

**RQ.N5. *Do users store notebooks with retrospective data?*** Displaying execution results is part of the literate programming aspect of notebooks. The support for rich media enhances the narratives and the writing of programs for people. Moreover, having partial cell results helps in checking the reproduction of a notebook by allowing the comparison of the cell outputs upon re-execution. However, some advocate that the results of notebook execution should be removed before committing to avoid noise in diffs (STALEY, 2017). Furthermore, Jupyter is also used as an IDE for general-purpose software development with the goal of extracting the produced code to scripts afterwards (KERY et al., 2018). We analyze the number of notebooks that have retrospective data and whether Jupyter is used as a literate programming tool by looking at the output formats (i.e., MIME types of cells’ outputs) in executed notebooks.

**RQ.N6. *How are notebooks executed?*** Jupyter allows users to execute cells in any order. While notebooks present the cells in a linear top-bottom narrative, a user may choose to execute the cells in a non-linear, arbitrary order. This ability departs from how most people expect to run code (GRUS, 2018; MUELLER, 2018; POMOGAJKO, 2015). Moreover, cells that appear at the beginning of notebooks may depend on cells that appear later, leading to additional issues for users that run them in the default top-down order (KOOP; PATEL, 2017). Figure 3.2 presents an unordered notebook and two re-executions of it following distinct execution orders: cell execution counter order and top-down order. In this example, the order that produces the same results is the one that follows the cell execution order. To quantify the prevalence of this practice, we identify notebooks that have cells in a non-linear order.

In addition to out-of-order cells, when Jupyter executes a code cell, the execution may change a state in the environment. It does not cause problems when users run cells only once and

In [5]: a = 2	In [2]: a = 2	In [1]: a = 2
In [4]: b = 3	In [1]: b = 3	In [2]: b = 3
In [7]: b + a	In [4]: b + a	In [3]: b + a
Out[7]: 2	Out[4]: 2	Out[3]: 5
In [6]: a = 1 b = 1	In [3]: a = 1 b = 1	In [4]: a = 1 b = 1
Original	Exec. Counter	Top-down

Figure 3.2: Original notebook and two executions that follow different orders.

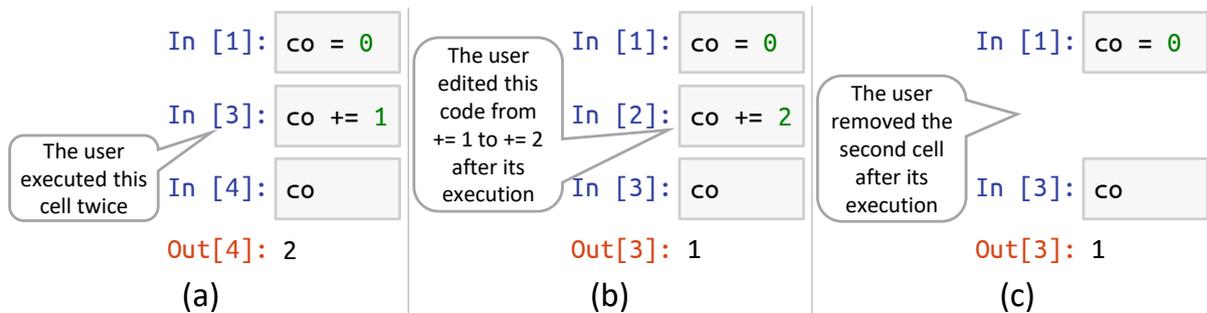


Figure 3.3: Three types of Hidden States: (a) Re-execution; (b) edited cell; (c) removed cell.

do not change previously executed cells. However, when the user runs the same cell multiple times, edits, or removes the cell code after executing it, the environment state may no longer represent the code definition, and this can lead to bugs and make debugging harder (GRUS, 2018; POMOGAJKO, 2015).

Figure 3.3 presents three examples of hidden states caused by these situations. Having a hidden state may make it impossible to reproduce the same results upon the re-execution of the notebook. In fact, the re-execution of these notebooks would produce results that differ from the ones in the output cell. Note that hidden states caused by cell re-execution or removal make the notebooks skip numbers in the execution counter sequence. Thus, in our analyses, we count how many execution counters skips there exist in the notebooks. Also, note that a removed or re-executed cell that causes a skip number does not necessarily produce a hidden state when it has code that does not change the environment. Hence, our measurement states the susceptibility of notebooks to have hidden states rather than confirming that they have them. Additionally, our analysis does not consider hidden states caused by edited cells that were not executed.

Table 3.1: Execution modes for the reproducibility experiments.

#	Mode	Environment	Execution order
1	Shared + Exec. Counter	Shared OS with conda and anaconda environments	Cell Execution Counter
2	Isolated + Exec. Counter	Isolated docker container with an anaconda environment	Cell Execution Counter
3	Isolated + Top-Down	Isolated docker container with an anaconda environment	Top-Down
4	Bloated + Exec. Counter	Bloated docker container with many dependencies installed	Cell Execution Counter
5	Bloated + Top-Down	Bloated docker container with many dependencies installed	Top-Down

We can only analyze the presence of skips and out-of-order cells in unambiguous execution order notebooks. We define *unambiguous execution order notebooks* as notebooks that have only one valid execution sequence. That is, they neither have cells with repeated execution counters, nor cells whose counter indicates that they are being executed. Note that this definition does not guarantee that the notebook outputs represent a single execution, but it is a close approximation with practical implications in our analyses.

Finally, the presence of non-executed code cells in the middle of the notebooks also hinders the reasoning about the execution. We analyze this issue by counting how many non-executed cells are in the notebooks and by comparing their positions with the position of executed ones.

**RQ.N7. How reproducible are notebooks?** Notebooks do not declare the versions of imported libraries (GRUS, 2018). The lack of version information may cause incompatibilities and prevent the execution of the notebook in environments that are different from the one in which the notebook was created. In Python, this issue can be addressed by defining dependencies in standard files: `setup.py`, `requirements.txt`, and `Pipfile`. We analyze how many notebooks belong to repositories with such files.

The existence of hidden states, out-of-order cells, hard-coded paths, and other bad practices also prevent the reproduction of notebooks. To assess the rate of reproducibility, we perform a reproducibility analysis of all unambiguous execution order Python notebooks. An unambiguous execution order notebook can have non-executed code cells in the middle. We ignore these cells since they do not have outputs.

**Execution modes.** In this analysis, we try to execute notebooks in five different modes to assess their reproducibility rate, as summarized in Table 3.1. We assess the rate by identifying notebooks that, when executed, lead to results that are the same as the results stored within the notebooks.

In the first execution mode, we executed the notebooks following the *cell execution counter order* in a *shared* OS environment with conda and anaconda environments to manage multiple Python installations and kernels. *Conda* is a package and environment management system that installs and manages the dependencies of packages. It allows multiple versions of Python to be installed with different dependencies. Conda was originally designed as part of *anaconda* (ANACONDA, 2018), which is a Python and R distribution that includes over 100 Scientific Packages, such as *numpy*, *scipy*, *matplotlib*, and other packages. Today, anaconda is a conda package that includes all these dependencies. In this work, we refer both to *conda environment* and *anaconda environment*. When we refer to conda environment, we consider an environment with only Python and Jupyter installed. When we refer to anaconda environment, we consider the environment that bundles the anaconda package and all of its dependencies. The decision on which environment to use was based on the availability of dependency declarations in the repositories. For repositories that declared dependencies, we used a conda environment and attempted to install the dependencies. For the other repositories, we used an anaconda environment with multiple pre-installed packages.

Since the first execution mode uses a shared OS environment, one execution can change system dependencies and affect the execution of other notebooks. Hence, we define four additional execution modes as an attempt to reduce the number of false negatives. In the second execution mode, we use docker containers to *isolate* the executions, and we run cells following the existing cell execution counter. In the third execution mode, we also use docker containers, but we run cells following the *top-down* order. In both these modes, we have anaconda environments installed in the containers. In the fourth and fifth execution modes, we also use docker containers, but we created *bloated* containers by attempting to install the maximum number of packages that we could install. The goal was to prevent notebooks from failing to reproduce due to the lack of dependencies. In the fourth execution mode, we executed notebooks following the existing cell execution counter, and in the fifth execution mode, we executed notebooks following the top-down order. Table 3.1 presents all the execution modes that we use in this chapter. We set a time limit of 5 minutes for the execution of each notebook.

**Normalizations.** In the first execution mode, we performed a direct character-by-character comparison of the output of a re-execution with the saved result to check the reproducibility results. However, this comparison can lead to false negatives due to small differences. For instance, the cell execution counter is part of the cell output. In the first analysis, every notebook with a skip would lead to a non-reproducible notebook. Similarly, insignificant deviations in number, date, and other object formats would lead to a non-reproducible notebook despite the

Table 3.2: Normalization Operations for Comparing Execution Results.

Operation	How	Reason
Encode	Encodes outputs into UTF-8.	Some notebooks were stored in a different encoding, leading to mismatches.
Execution Counter	Removes the execution counter from outputs.	Skips in the execution counter lead to mismatches.
Stream	Combines print sequences into a single output element.	Different versions of Jupyter/IPython behave differently, leading to mismatches on print statements.
Dictionary	Alphabetically sorts dictionary keys and set elements.	The order of these elements does not matter, but the textual comparison fails for unordered objects.
Dataframe	Removes HTML representation of <i>pandas</i> dataframes, keeping only textual representations.	<i>Pandas</i> outputs both text representations and HTML representations of the same dataframes, but the HTML representation has changed over time for styling reasons, leading to mismatches.
Exception Path	Removes paths from exceptions.	Python exceptions show the file path, leading to mismatches in different machines.
Deprecation	Removes deprecation warnings.	Deprecation warnings in new versions of libraries lead to mismatches.
White space	Transforms all kinds of white spaces into a single space.	The representation of line breaks and other white space characters changes from system to system, leading to mismatches.
Decimal	Cuts numbers at the second decimal place.	Small variations in float precision lead to mismatches.
Date	Replaces dates by 1970-01-01T.	Running a notebook that outputs the current date at two different dates would result in a mismatch.
Time	Replaces time by 00:00:00.	Running a notebook that outputs the current time at two different times would result in a mismatch.
Memory	Replaces numbers that start with 0x by 0x0000000.	Python objects often indicate their position in the memory on print statements. Since every execution puts the object in a different position, keeping the original number leads to a mismatch.
Image	Removes images from outputs.	It is hard to compare images, and very small changes in image generation lead to different results.

notebook producing a similar result that is semantically the same. To avoid this problem, we normalize the notebook outputs in the latter four execution modes. The normalization operations we applied are presented in Table 3.2. We apply these operations in the same sequence they appear in Table 3.2. Hence, before we apply the stream normalization, we apply both the encode normalization and the execution counter normalization. Thus, having skips in the execution counter or insignificant deviations in formats does not lead to non-reproducible notebooks, reducing the number of false negatives. We indicate the reproducibility rate for each normalization. Note that the Image normalization is expected to have the highest rate, as it includes all the previous normalization operations. On the other hand, the image normalization is expected to cause false positives, as it strips images out of the notebooks.

**Mining Relationships between Notebook Features and Reproducibility.** To gain deeper insights into factors that influence reproducibility, we mined association rules (AGRAWAL;

SRIKANT, 1994) that relate specific notebook features to both success and failure to reproduce notebooks.

Our data mining strategy was conservative by nature – we used a low absolute support threshold of 100 transactions and did not use a confidence threshold. This is important to avoid hindering infrequent but relevant rules. Then, we mined for rules with size two (one feature in the antecedent and one in the consequent) using the Apriori algorithm (AGRAWAL; SRIKANT, 1994) provided by package *arules* in R. Next, we sorted the obtained rules descending by lift and fixed features related to reproducibility in the consequent. Finally, we observed which features appeared in the antecedent with lift significantly higher or lower than one.

### 3.3.2 Data Acquisition and Preprocessing

We used the GitHub API to find repositories created between January 1st, 2013 and April 16th, 2018 that had a file with “Jupyter Notebook” as identified language. This query returned 265,888 repositories with 1,450,071 notebooks. We did not collect checkpoint notebooks stored in `.ipynb_checkpoints` directories. Most repositories (60.09%) have 2 or fewer notebooks. Only 12.42% of the repositories have 10 or more notebooks. However, 61.45% of the notebooks belong to repositories with 10 or more notebooks.

On July 22nd, 2020, we queried GitHub again to obtain the number of stars and forks of these repositories. Some repositories were removed from GitHub in this interval. Thus, we removed them from our analyses as well, resulting in 1,274,872 notebooks from 235,643 repositories.

After collecting the repositories, we excluded invalid notebook files, empty notebooks, empty repositories, and repositories that we lost access between the query moment and the analysis moment, resulting in 1,251,074 valid notebooks from 234,729 repositories. From this result, we also excluded 226,805 (18.13%) duplicated notebooks. The goal was to reduce the bias towards forks and notebook copies (KALLIAMVAKOU et al., 2014). We detected these notebooks by calculating the SHA1 hashes from cell sources and output formats. We did not use the output results or other metadata when we calculated the hashes to be able to detect notebooks that only had distinct prospective data as duplicates. This resulted in 1,024,269 unique notebooks for the analyses.

### 3.3.3 Popular Notebooks Selection

From the set of unique notebooks, we extracted a set of popular notebooks representing mature notebooks to use it as a baseline for the analyses. This selection reasoning is that those mature notebooks should condense the most quality and reproducibility characteristics as they have received the most attention from users. In fact, we observed that popular notebooks attain more quality features, such as having more Markdown cells, fewer issues with titles, and less out-of-order cells and skips. As we discuss in Section 3.4.3, popular notebooks are more reproducible than the overall group as well. They are 31.04% more likely to execute until the end, 84.83% more likely to reproduce the same results without normalizations, and 41.34% more likely to reproduce the same results after all normalizations.

For selecting the set of popular notebooks, we approximated the popularity of the notebooks based on the popularity of their repositories. Hence, we first assigned to each notebook the number of stars and forks of the repositories containing them. We then computed a popularity score ( $s$ ) that consists of the harmonic mean of stars and forks of the notebooks. From this, we removed notebooks with zero as the popularity score and obtained top outliers of the remaining notebooks as follows:

$$\begin{aligned}
 s &\geq 1.5 \times IQR \\
 &\geq 1.5 \times (Q3 - Q1) \\
 &\geq 33.331
 \end{aligned}
 \tag{3.1}$$

This selection resulted in a popular set of 38,063 notebooks, which corresponds to 3.72% of the unique notebooks.

### 3.3.4 Sampling

To get more insight into the context in which good and bad practices are applied, we systematically extracted a sample of real notebooks from the complete set of unique notebooks. With this sample, we can observe characteristics that we could not extract automatically from the notebooks and use real notebooks as examples. We used Cochran's sample size formula (ISRAEL, 1992) to calculate our population's sample size. Assuming a maximum variability ( $p = 0.5$ ), and desiring a confidence level of 90% and  $\pm 10\%$  precision range, we calculated the sample size as 68.05 notebooks. Hence, we randomly selected 69 notebooks as our sample.

As a sanity check, we compared 75 metrics with percentages related to the broad set of unique notebooks that we report throughout this chapter with their sample counterparts. We found that only 11 metrics deviate from the  $\pm 10\%$  range when using sub-group selections. However, all of them are calculated based on sub-groups that are not representative enough in the sample (e.g., a percentage over the number of “Untitled” notebooks, which corresponds to only 1.93% of the notebooks). When we do not consider sub-group selections (i.e., we compare the percentages over the total number of unique notebooks or sampled notebooks), all analyzed metrics are within the  $\pm 10\%$  range.

In the sample, we found 31 notebooks associated with courses, such as tutorials, class assignments, or course exercises. We also found ten academic notebooks related to papers, dissertations, theses, and capstone projects. Ten notebooks had analysis for tasks not related to education – although some are related to writing blog posts. Nine notebooks were related to personal practicing, such as solving book exercises or exploring new things. Five notebooks described how to use other tools and libraries. Three notebooks are related to books. A notebook is part of a presentation. Given the number of notebooks related to education, understanding and supporting notebooks may greatly impact educational projects.

Given that most notebooks belong to educational projects, we analyzed to which area they belong the most. We counted 28 notebooks related to data exploration, using simple *pandas* functions and plots. Ten notebooks were related to machine learning, with libraries such as *keras*, *sklearn*, and *lasagne*. Additionally, six notebooks were related to data mining, with algorithms for clustering and building decision trees. Five notebooks were related to data cleaning, to transform a data format into another. While most notebooks were data-centric, we also found notebooks related to other areas: programming (four notebooks), databases (three notebooks), math problems (three notebooks), algorithms (three notebooks), computer vision (two notebooks), games (one notebook), computer graphics (one notebook), and physics (one notebook). The last two notebooks only had markdown cells with tasks for students. These results indicate that most notebooks are data-centric analyses.

Despite most notebooks sharing areas and being data-centric, most of them use different datasets for their analyses. Some of them use toy datasets available in existing libraries – others use real data from many different contexts.

### 3.3.5 Corpus

We analyzed the declared programming languages of all unique notebooks. Figure 3.4 presents, in the log scale, the 15 most declared programming languages we found. Python is by far the

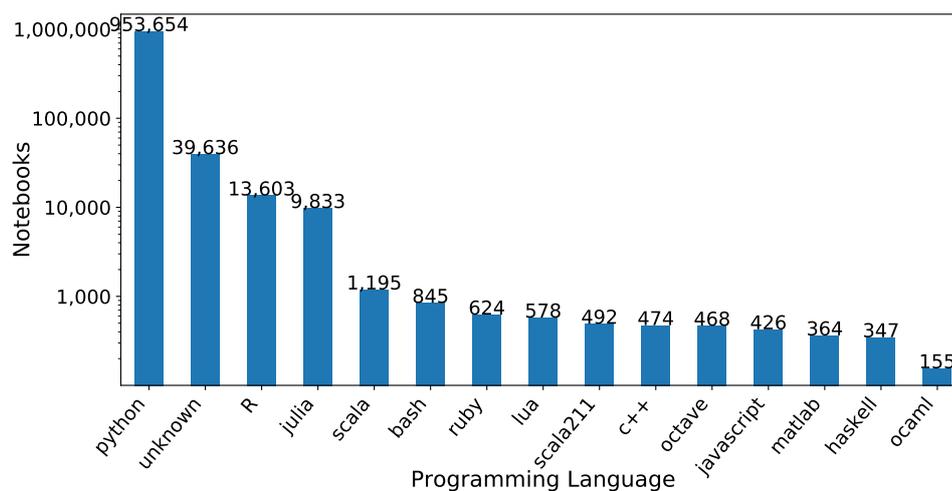


Figure 3.4: Top 15 most declared programming languages. Notebooks axis in logarithmic scale.

most used programming language, corresponding to 93.11% of the notebooks. It is followed by R (1.33%) and Julia (0.96%). The popular group has slightly more Julia notebooks (1.05%) than R notebooks (0.99%), but Python still represents most of the notebooks. All notebooks in the sample declare Python as their programming language. However, two notebooks could declare any or no programming language, as they do not have code cells. Moreover, two notebooks are composed mostly of *cell magics* with SQL queries. We also found five Python notebooks in the sample that used bang expressions to invoke shell commands.

Due to the interactive nature of notebooks, most programming languages are scripting languages. Nonetheless, Jupyter is also used for compiled languages such as C++ and Haskell. A total of 39,636 unique notebooks do not declare a programming language, and 30,953 of them use *nbformat* lower than 4, which predates the release of the language-agnostic Jupyter. Although this is a strong indication that these notebooks also use Python, we opted for removing them from Python-specific analyses.

Since most notebooks contain Python code (953,654) and questions RQ.N3, RQ.N4, and RQ.N7 require language-specific analyses, we focus on Python notebooks to answer these questions. We extracted declared versions and cells with metadata from Python notebooks, and we used the Python AST to extract Python constructs and imported modules. The most used version is Python 2.7, which corresponds to 36.38% of the Python notebooks. However, by combining minor releases, Python 3 surpassed Python 2. In fact, 63.53% of the Python notebooks use Python 3. The remaining did not declare a version. For RQ.N3 and RQ.N4, we used only valid Python notebooks (i.e., notebooks with a valid Python syntax in all code cells). Valid Python notebooks correspond to 886,668 (92.98%) notebooks. For RQ.N7, we did not have this re-

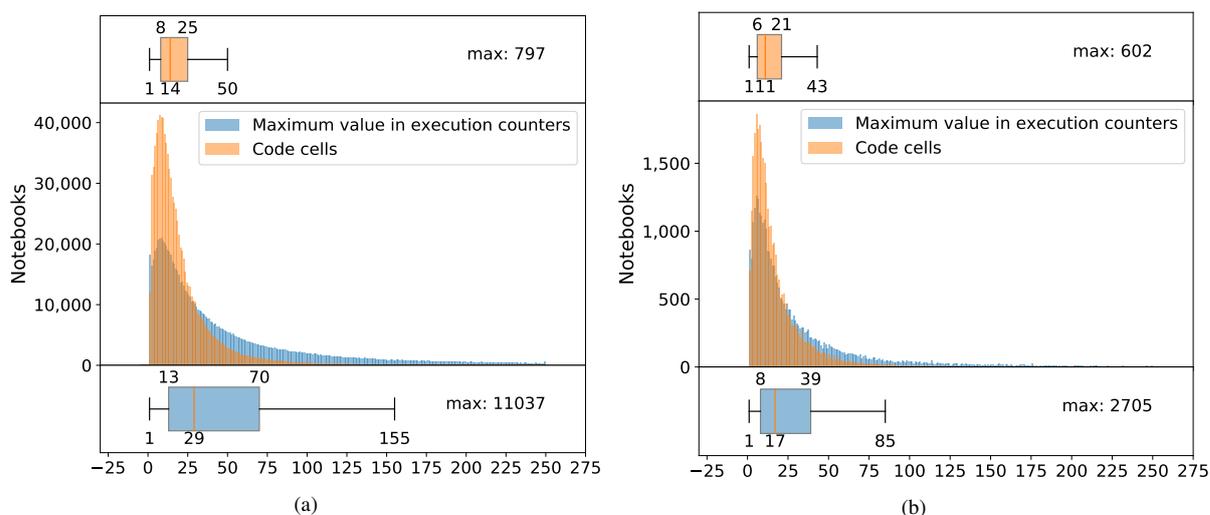


Figure 3.5: Distribution of code cells and maximum execution counter for overall group (a) and popular group (b).

striction, because we ran only executed cells of Python notebooks with unambiguous execution order, which correspond to 753,405 (79.00%) notebooks.

In addition to these restrictions, we analyzed only executed notebooks for RQ.N5 and RQ.N6, corresponding to 932,382 (91.03%) notebooks. Figure 3.5 presents the distributions of code cells and maximum execution counter value by executed notebooks for the overall group (a) and popular group (b). Both distributions concentrate at the beginning of their histograms, indicating that notebooks are relatively small both in the number of code cells and in the number of maximum execution counter, compared to the size they can get. However, the mismatch between the number of code cells and the maximum execution counter number – which can be observed both by comparing the median or the visual representation – indicate that notebooks have more executions than cells (i.e., they necessarily have skips). Popular notebooks are even smaller (shorter medians), and the difference between the maximum execution counter and the number of code cells is smaller than that of the overall group as well. It indicates that they have fewer skips. Their execution counter is closer to the code definition.

Figure 3.6 summarizes the corpus of this chapter. The percentages reported in each research question’s analysis refer to the number presented in this corpus unless stated otherwise. While we indicate the number of samples in each group in this figure, we still discuss all the samples in the analyses, since some restrictions do not hold for manual qualitative analyses. For instance, to assess the execution order (RQ.N6), we do not need to attain to notebooks with unambiguous execution order, since we are not using automatic analyses. Instead, we can try to understand the order in ambiguous notebooks based on the context.

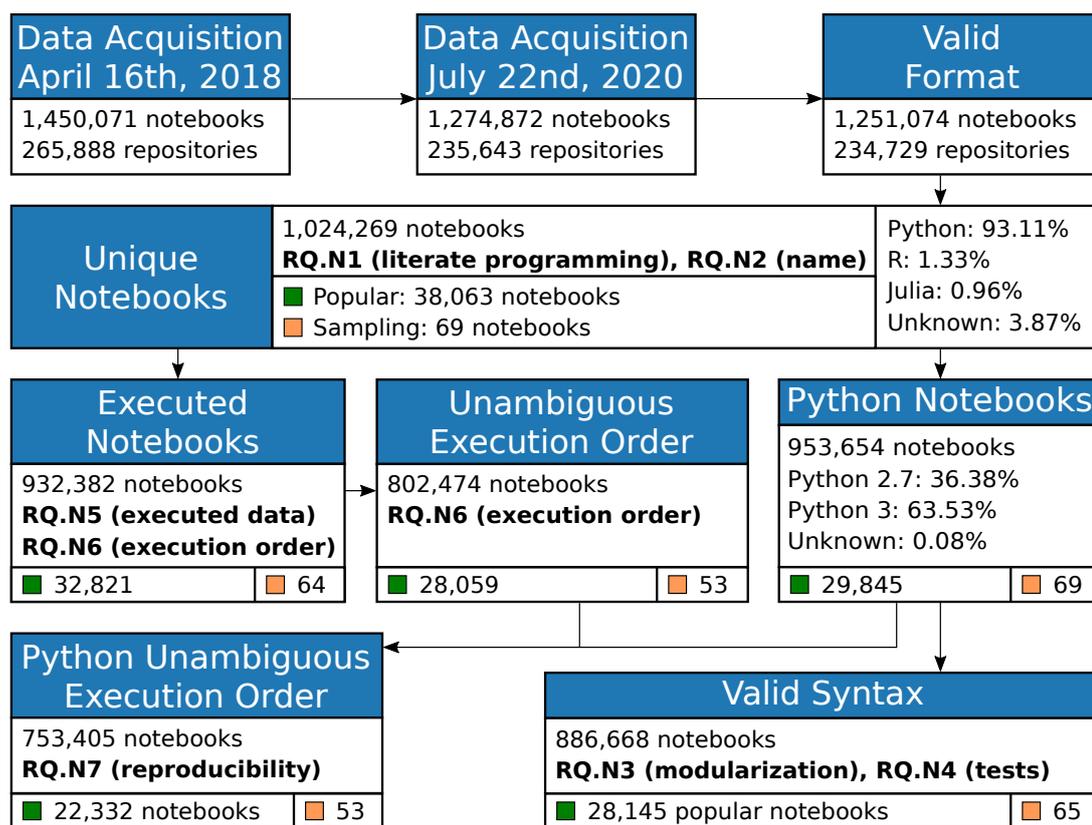


Figure 3.6: Notebook corpus and its partitions used in the analyses.

## 3.4 Results

In this section, we present the results we collected to answer the research questions of this chapter. We opted to remove the full discussions of the research questions related to prospective data (RQ.N1– RQ.N4) to keep the chapter more concise and closer to the scope of the thesis. Table 3.3 presents a summary of the findings for these questions. A brief discussion of all research questions can be found in the paper we published in the International Conference on Mining Software Repositories (PIMENTEL et al., 2019b), and the complete discussion can be found in the extended paper that was accepted in the Empirical Software Engineering (PIMENTEL et al., 2021). The following subsections discuss the research questions related to retrospective data (RQ.N5– RQ.N7).

### 3.4.1 RQ.N5. Do users store notebooks with retrospective data?

**Outputs.** As stated in Section 3.3.2, we collected 932,382 executed notebooks, which corresponds to 91.03% of the unique notebooks. These notebooks have retrospective data.

Table 3.3: Results of research questions related to prospective data.

RQ	Results
RQ.N1. How do notebooks use literate programming features?	<p><b>Answer:</b> Most notebooks have Markdown cells. Moreover, Markdown cells correspond to almost one-fourth of the cells. On the other hand, the text is often short, and the most used elements are simple headers and paragraphs, despite the possibility of displaying lists, images, links, and other formatted elements. Their position indicates that users give more attention to the beginning of notebooks. In the sample, we observed that notebooks use headers to separate sections and describe code cells. In addition to describing code cells, notebooks also use Markdown to describe the problem/goal they are tackling, describe tasks, and conclude the document.</p> <p><b>Implications:</b> The small size and usage of few Markdown elements potentially compromise the understandability of the notebook. Additionally, Markdown could provide descriptions on how to reproduce the notebook, such as indicating libraries to install or the execution order. Hence, reproducing the last cells of an average notebook that does not have Markdown cells may represent a challenge.</p>
RQ.N2. How are notebooks named?	<p><b>Answer:</b> Most users seem to change the default name in the titles of their committed notebooks and use meaningful but short names. On the other hand, many users do not seem to be concerned about OS-based restrictions and conventions in naming files. In the sample, we observed that some repositories define a sequence of execution for the notebooks.</p> <p><b>Implications:</b> Disregarding OS-based naming constraints may hamper the reproducibility when using other operating systems. The sequencing in the naming scheme is important both for the reproducibility (e.g., executing a notebook that depends on data generated by a previous notebook in the sequence) and for the literate aspect of the notebooks (e.g., executing a notebook that deepens on the explanation of a concept that was introduced in previous notebooks).</p>
RQ.N3. How do notebooks use modules, functions, and classes?	<p><b>Answer:</b> On the one hand, users seem to create functions in notebooks that have more complex code with control flow constructs. On the other hand, users do not seem to extract functions to local modules, given the fewer number of notebooks with local modules. Class definitions are indeed rare, but it may be a consequence of the multi-paradigm design of Python.</p> <p><b>Implications:</b> While defining functions and classes inside notebooks achieves the benefits of reusability and abstraction, these benefits are limited to internal use of the notebook. Local modules could be better explored to extend the reusability to other notebooks and scripts, and reduce the size of code cells in notebooks. However, keeping the code inside the notebook can be good for reproducibility, as it allows users to share only the notebook file with all code.</p>
RQ.N4. How are notebooks tested?	<p><b>Answer:</b> Very few notebooks import testing modules. However, we observed in the sampled notebooks that some repositories attempt to test code related to the notebook outside the notebook environment.</p> <p><b>Implications:</b> There is an opportunity for improving tests on notebooks. An appropriate test suite is important for assuring the reproducibility in other environments. However, for notebook code that is based on data exploration, the existing tools are not sufficient and too intrusive. It also opens the opportunity for proposing testing approaches for notebooks.</p>

Table 3.4: Output formats in cells and notebooks. Note that a cell can have multiple output formats, thus, the percentages add up to more than 100%.

Format	Overall		Popular	
	% of cells with output	% of executed notebooks	% of cells with output	% of executed notebooks
Text	68.11%	82.00%	58.48%	67.21%
Stream	36.08%	70.47%	44.32%	65.28%
Image	22.67%	51.69%	19.84%	43.61%
HTML/JS	16.23%	36.92%	12.08%	26.98%
Error	2.28%	14.86%	1.15%	6.92%
Formatted	1.22%	1.91%	1.31%	1.75%
Extension	0.44%	1.56%	0.32%	0.97%
PDF	0.08%	0.12%	0.08%	0.11%

Among the executed notebooks, 54.31% of the code cells had an output, and 96.20% of the notebooks had at least one cell with an output. Despite having fewer code cells, popular notebooks have more code cells with an output, proportionally (59.66%).

Table 3.4 presents the percentage of cells and notebooks with each output format for both the set of executed notebooks and executed popular notebooks. Note that a cell can have multiple output formats. Thus, the percentages add up to more than 100%. The same happens for notebooks. In this table, *Text* represents the textual output of cells, and *Stream* represents the output of print statements and exceptions. *Image* represents PNG, JPEG, and SVG formats, which are the default image formats supported by Jupyter. *Formatted* represents *Markdown* and  $\LaTeX$  formats. Finally, *Extension* combines all extension-specific formats. The most common extension formats are Jupyter Widgets, *plotly*, and *bokeh* formats. Very few notebooks use the extension formats. Note in this table that most executed notebooks have outputs in cells. Note also that despite having proportionally more executed cells with outputs, the popular group has a smaller percentage of all output formats. It might indicate that cells in the overall group tend to have multiple outputs at once, while cells in the popular group tend to be more focused on a smaller number of outputs.

**Sampled Notebooks.** In the 69 sampled notebooks, only five notebooks (7.25%) do not have execution data. Two of them are the notebooks mentioned before that only have tasks descriptions in the Markdown. In the notebooks with retrospective data, we found eight types of output data: 53.62% of the notebooks with results on cell outputs, 5.80% with accidental results on cell outputs (e.g., functions that configure *matplotlib* plots returning objects at the end of cells whose main goal is to display plots), 72.46% with stream outputs, 46.38% with images, 43.48% with tables, 21.74% with warnings, 14.49% with exceptions, and 7.25% with interactive components that did not load without re-executing the notebooks. Some of these components use

```
[15]: just_dummies2 = pd.get_dummies(nycmodel1['zipcodeHour1'])
```

```
[16]: just_dummies2.shape
```

```
[16]: (613548, 4673)
```

```
[17]: just_dummies2.head()
```

	10001_0	10001_1	10001_10	...	11451_7	11451_8	11451_9
0	1.0	0.0	0.0	...	0.0	0.0	0.0
1	0.0	0.0	0.0	...	0.0	0.0	0.0
2	0.0	0.0	0.0	...	0.0	0.0	0.0
3	0.0	0.0	0.0	...	0.0	0.0	0.0
4	0.0	0.0	0.0	...	0.0	0.0	0.0

```
[5 rows x 4673 columns]
```

```
[19]: just_dummies2.to_csv('just_dummies2.csv', sep='\t',
    ↪encoding='utf-8')
```

```
-----
KeyboardInterrupt                                Traceback (most recent call last)
<ipython-input-19-8a564964362c> in <module>()
      1 os.chdir("/Users/binfang/Documents/NYCDSA/project/Project_5/
    ↪data/processed_data")
----> 2 just_dummies2.to_csv('just_dummies2.csv', sep='\t',
    ↪encoding='utf-8')
```

Available at:



Figure 3.7: Snippet of `pythoncode/improvedlm.ipynb` from the GitHub repository `poorbaby/Predict-New-York-Taxi-Demand`.

HTML and JS in the output, while others use extensions. Finally, we also identified that 27.54% of the notebooks write files in addition to the usual notebook output.

Note that these categories are somewhat different than the ones we reported in Table 3.4. This mainly happens because we analyzed the outputs in the sampled notebooks using the Jupyter Lab interface instead of reading their JSON representations. It leads to two major consequences. First, as humans, it is easier for us to visually identify that a cell has a table than to identify that a cell outputs HTML to display the table – everything is HTML in the Jupyter Lab interface. Similarly, we can easily identify whether a stream output is a warning message or just the result of a print statement. Second, in some situations, we are only able to identify one output type, despite a given cell generating at the same time multiple outputs (e.g., a *pandas* table has both an HTML representation and a text representation). This happens because Jupyter only displays the most appropriate for the application in such situations but stores both results in the notebook file.

Figure 3.7 presents a snippet of one of the sampled notebooks that has a cell with no output (In [15]), and cells that output a text string at Out [16], an HTML table at Out [17], and an error at Out [19]. It is common to find notebooks containing multiple output formats.

**RQ.N5.** *Do users store notebooks with retrospective data?*

**Answer:** Most notebooks store cells with outputs.

**Implications:** This result fosters reproducibility. Knowing the expected output allows users to re-run notebooks and check if they reproduce the results.

### 3.4.2 RQ.N6. How are notebooks executed?

**Executed Notebooks.** Among the 932,382 executed notebooks, 21.11% had non-executed code cells, and 62.14% had empty cells. Figure 3.8 presents the distribution of code cells in the notebooks. Note that the percentage of executed code cells drops towards the bottom of notebooks, while the percentage of non-executed and empty cells grows. While 59.15% of executed notebooks finish with empty cells, only 11.35% of executed notebooks have empty cells among non-empty ones. Popular notebooks are 38.43% less likely to have non-executed cells, 38.80% less likely to have empty cells in the end, and 56.14% less likely to have empty cells among non-empty ones.

**Unambiguous Execution Order.** We collected 802,474 notebooks with *unambiguous execution order* (i.e., the ones that neither have repeated values in execution counters nor executing cells, marked with an asterisk). This number corresponds to 86.07% of the executed notebooks. Among the notebooks with unambiguous execution order, 36.45% have out-of-order cells. The percentage of unambiguous notebooks in the popular group is very close (85.49%), but only 22.37% of them have out-of-order cells.

By following the execution counters' sequence in unambiguous execution order notebooks, we counted how many skips occurred. Since skips represent cell executions without explicit

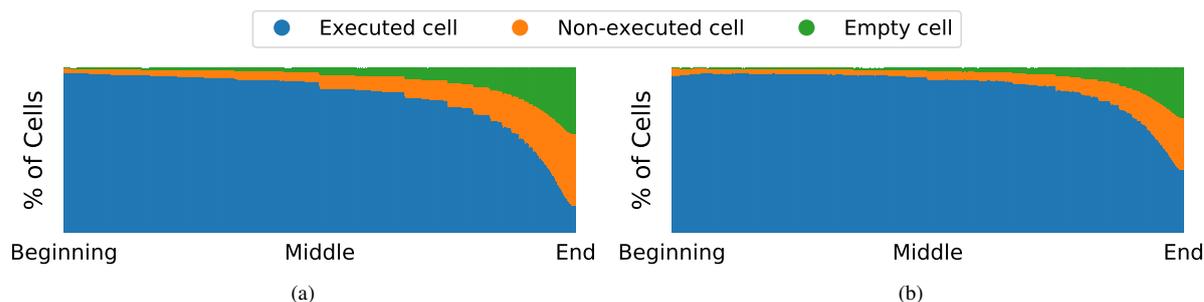


Figure 3.8: Distribution of code cells in executed notebooks (a) and popular notebooks (b).

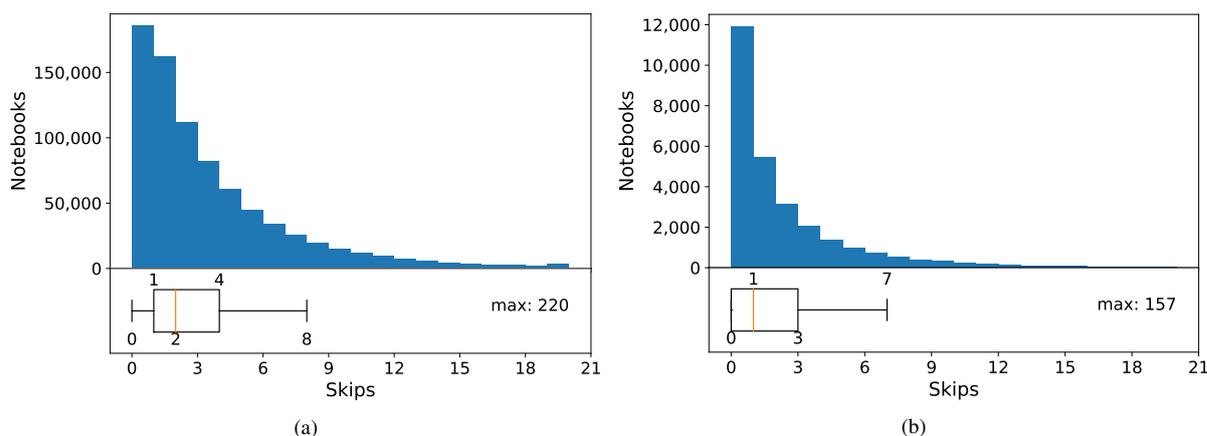


Figure 3.9: Distribution of skips in notebooks with unambiguous execution order (a) and popular notebooks (b).

definitions, they may indicate the presence of hidden states. Figure 3.9 presents the distribution of skips by notebooks. 76.88% of unambiguous execution order notebooks have at least one skip. A skip contains 12.83 executions on average. By considering only skips in the middle (i.e., excluding skips in the first cell), the percentage of notebooks with skips drops to 66.15%. Additionally, the average of skipped executions drops to 10.33. As expected, all these numbers drop as well for popular notebooks: 57.54% of them have skips, and 47.84% of them have skips in the middle. A skip contains 9.84 executions on average, or 8.31 executions when we only consider skips in the middle.

**Sampled Notebooks.** Among the 69 sampled notebooks, we found 28 unordered notebooks for exploratory reasons (40.58%), such as updating plots, reloading data, or changing the algorithm. Among them, 19 notebooks had cells defining names (i.e., variables and functions) executed after the cells that use them, leading to non-reproducible notebooks. Moreover, 15.94% of the notebooks had ambiguous execution order due to the repetition of cell numbers, making it hard to execute them. In some cases, the repetition occurred following the top-down order, indicating an attempt to re-execute the notebook in a new session that did not run all cells. In other cases, the notebook had results from two separate sessions, with one of them executing cells at the beginning and the other executing cells at the end. Some unordered notebooks had repeated cell numbers spread throughout the notebook, making it hard to understand the desired execution intention.

We also observed that 8.70% of the sampled notebooks have cells that were noticeably edited after their execution (e.g., cells with output that could not be generated by that cell code), and 36.23% of the notebooks have non-executed code cells, making it hard to decide which

```
[10]: engine = create_engine('postgresql://postgres:root@localhost:5432/
↳'+dbname)

cancellations.to_sql("cancellations", engine, if_exists = "replace")
operations.to_sql("operations", engine, if_exists = "replace")
airports.to_sql("airports", engine, if_exists = "replace")
```

Join airport\_cancellations.csv and airports.csv into one table

```
[ ]:
```

Query the database for our initial data

```
[38]: cur = conn.cursor()
cur.execute("""SELECT * FROM age""")
ap = cur.fetchall()
print ap
```

```
File "<ipython-input-38-d34049c3d36a>", line 4
  print ap
    ^
SyntaxError: Missing parentheses in call to 'print'
```

Available at:



## 1.2 What are the risks and assumptions of our data?

### Part 2: Exploratory Data Analysis

#### 2.1 Plot and Describe the Data

```
[ ]: ap.head()
ap.describe()
```

Figure 3.10: Snippet of pparker-roach/project\_7-SANDBOX.ipynb from the GitHub repository mohsseha/DSI-BOS-students.

cells should be executed. Nonetheless, 32.00% of these notebooks with non-executed cells had them at the end of the notebook, indicating that the users stopped executing the cells at a given moment. Additionally, 16.00% of these notebooks have non-executed code cells with only code comments. We also found that a notebook had an incomplete code cell that was not executed, and a notebook had non-executed cells after an exception, which might have prevented the user from running the end of the notebook using the *Run all cells* option.

Figure 3.10 presents a snippet of one of the sampled notebooks. This notebook has an empty cell between two Markdown cells, a skip in the execution count, and non-executed code cells in the end. While the skip in the snippet is from In [10] to In [38], the actual skip in the notebook is from In [17], as the notebook has the cells in the wrong order. Having cells in the wrong order is also a source of hidden states in this case: the cell In [12] appears at the beginning of the notebook, but it redefines the variables `cancellations`, `operations`, and `airports` used in the In [10], presented in the snippet. Hence, executing this notebook following the cell execution counter would fail.

**RQ.N6.** *How are notebooks executed?*

**Answer:** Many unambiguous execution order notebooks have non-executed code cells, out-of-order cells, and skips in the execution counter. All these characteristics hinder the reasoning about execution states. The number of notebooks with skips and the average size of skips drop when we exclude skips at the beginning of the notebooks. A possible cause for these skips happening only at the beginning of a notebook is the re-execution of all of its cells without restarting the kernel.

**Implications:** There is an opportunity for proposing approaches that measure non-executed code cell, out-of-order cells, and skips as code smells in notebooks, i.e., structures in the code that violate design principles and can negatively impact quality (GAROUSI; KÜÇÜK, 2018). Fortunately, most of these code smells are easily fixable by restarting the kernel and executing all cells again before committing. Nonetheless, such an approach could detect out-of-order cells by looking not only to cell numbers but also to variable usages occurring before their definition.

### 3.4.3 RQ.N7. How reproducible are notebooks?

**Handling Dependencies.** To answer RQ.N7, we conducted a reproducibility study in which we attempted to execute all 753,405 Python notebooks with unambiguous execution order. Among these, 94,183 (12.50%) belong to repositories that declared module dependencies (which corresponds to 8.78% of the repositories that have Python notebooks with unambiguous execution order). Proportionally, a higher percentage of popular notebooks belong to repositories that declared dependencies (21.77%), suggesting that popular notebooks have more intention of providing directions for their reproducibility. These repositories correspond to 24.63% of the popular repositories that have Python notebooks with unambiguous execution order.

Among repositories with dependencies, 79.85% use `requirements.txt`, while 45.62% use `setup.py`. Many of these repositories (26.09%) have both `setup.py` files and `requirements.txt` files. Moreover, some repositories even have more than one of these files. In addition to these files, we found 865 notebooks that belong to repositories with `Pipfile`.

Popular notebooks are 7.80% less likely to have `requirements.txt` files, 22.20% more likely to have `setup.py` files, 13.71% more likely to have both, and 50.77% less likely to use `Pipfile`. Using more `setup.py` and less `requirements.txt` may indicate that popular

notebooks are part of repositories meant to be redistributed and used together with other projects (e.g., libraries) – in opposite to repositories that define the complete Python environment for their standalone execution. The reason they use less `Pipfile` may be related to their age and the time they needed to become popular, as `Pipfile` is a much more recent system.

Not all dependency declarations are valid. In the first execution mode (shared + execution counter), we attempted to install the dependencies for these notebooks in conda environments. However, the dependencies of 59.30% of the notebooks failed to install. To install the dependencies, we first installed all the `setup.py` files in the repository. Then, we installed the `requirements.txt` files. Finally, we installed the `Pipfile` files. The failure rates for these files were 65.53%, 57.21%, and 60.69%, respectively.

The failure rate for the installation of `requirements.txt` was lower than the other formats. While the `requirements.txt` is a declarative format in which the module version is pinned, the `setup.py` is a generic Python script that supports any flexible installation code. Thus, `setup.py` is more susceptible to errors. In comparison to `Pipfile`, `requirements.txt` is a well-established format that has been used for many years. `Pipfile`, on the other hand, was introduced less than four years ago, and its specification still goes through constant revisions.

The failure rate of `setup.py` was about the same for popular notebooks (64.92%). However, `requirements.txt` and `Pipfile` were less likely to fail: 47.42%, and 40.91%, respectively. The reason it happened may be related to the intention of the users when creating these files. Usually, Python developers create `setup.py` to install libraries and command-line tools (PYPA, 2020). This intention does not change according to the repository popularity. However, `requirements.txt` and `Pipfile` have the intention of either describing the dependencies of a complete Python environment or describing the dependencies of an application. We suppose popular repositories may design these files describing only the project dependencies to allow other users to use it, while non-popular repositories may use the `pip freeze` command to describe all the environment dependencies for a `requirements.txt` file and not face issues from other users.

Among the reasons for installation errors, we identified that 29.17% have files that require other unavailable files (e.g., sub-requirements and downloads from unavailable servers), 29.17% have malformed files (i.e., wrong syntax or conflicting dependencies), 25.59% have files that require a previous installation of Python packages (e.g., `setup.py` requires Cython to compile and build a package), 19.69% require external tools (e.g., compilers and libraries), 8.43% have files designed for other systems (e.g., Raspberry Pi and Windows), and 0.98% have

dependencies that do not support the declared Python version (e.g., the repository has a Python 2 notebook, but the `setup.py` requires a module that dropped support to Python 2 and did not pin the module version). Popular notebooks have fewer errors related to unavailable files (20.28%), malformed files (20.28%), being designed for other systems (5.94%), or having dependencies that do not support the Python version (0.10%), but more errors related to requiring a previous installation of a Python dependency (33.87%) or an external tool (25.09%). It is expected since there was no standard way to define dependencies during the development of most of these notebooks. The specification for Python build system requirements was proposed in May 2016 and implemented in March 2017 (CANNON; SMITH; STUFFT, 2016).

We were able to install the dependencies for 40.70% of the notebooks. In addition to these notebooks, we prepared anaconda environments for the notebooks that did not declare dependencies (87.48% of them). Unlike previous conda environments, an anaconda environment comes with a comprehensive set of scientific Python packages, such as *numpy*, *matplotlib*, and *pandas*. Combining both the set of notebooks for which we were able to install the dependencies and the set of notebooks that did not declare dependencies, we had 697,398 notebooks on our first execution mode. The installation success rate for popular notebooks was close: 41.38%.

**Isolating Executions and Exploring Different Execution Orders.** While the first execution mode used conda environments to isolate the dependency installation, it did not isolate the interactions between notebooks and the OS. Consequently, one execution could interfere with another by changing state in the *shared* OS. To address this limitation, we run notebooks inside *isolated* docker containers. Additionally, the first execution mode only executed notebooks by following the order of cell execution counters, which can lead to false negatives with respect to reproducibility assessment. In the other modes, we follow both the cell execution counter and the top-down order.

We also prepared *bloated* docker images in which we attempted to install all modules imported by all Python notebooks that we collected. As expected, many installations failed and we left them out of the image. Nonetheless, most popular modules, which were imported by more than 2,000 notebooks in our corpus, were successfully installed. The only exceptions were *GraphLab* (imported by 11,092 notebooks), *PyTorch* (imported as `torch` by 7,745 notebooks and as `torchvision` by 3,433 notebooks), *gensim* (imported by 7,174), and *GeoPandas* (imported by 3,350). *GraphLab* requires a license to use. *PyTorch* did not work in our environment. *GeoPandas* and *gensim* had binary dependencies that we could not install. In addition to these dependencies, we also installed common tools, compilers, and interpreters in all docker containers to reduce the number of failures due to the absence of external tools.

Table 3.5: Association rules related to timeout

Mode	Antecedent	Consequent	Support	Confidence	Lift
Isolated + Exec. Counter	unittests	timeout	0.09%	28.80%	11.56
Isolated + Exec. Counter	raise	timeout	0.17%	6.79%	2.73
Isolated + Exec. Counter	while	timeout	0.32%	4.21%	1.69
Isolated + Top-Down	unittests	timeout	0.11%	35.26%	9.81
Isolated + Top-Down	raise	timeout	0.21%	8.50%	2.37
Isolated + Top-Down	while	timeout	0.45%	5.86%	1.63

**Notebook Executions.** We report five execution modes by alternating both the execution order and the environment. In the first one (shared + execution counter), we executed notebooks following the cell execution counter in conda and anaconda environments installed in a shared system. The second one (isolated + execution counter) uses the anaconda docker image and runs notebooks following the cell execution counter. The third one (isolated + top-down) also uses the anaconda docker image, but we execute the cells following the top-down execution order. In the fourth one (bloated + execution counter), we use the bloated docker image to execute notebooks following the cell execution counter. Finally, in the fifth mode (bloated + top-down), we also use the bloated docker image to execute in the top-down order. Due to time constraints, we could not install the dependencies from the dependency files in the docker containers. Thus, we restricted the executions in the isolated modes only to notebooks in repositories that did not have dependency files. Additionally, we executed all notebooks in bloated modes without installing specific packages from dependency files of each repository.

In our experiments, many notebooks failed to execute all cells. Some failed because their execution exceeded a time limit of 5 minutes, while others failed due to an exception. Figure 3.11 presents the percentage of notebooks that failed due to timeout, in addition to the 10 most common exceptions the notebooks presented in each execution mode of our assessment.

By mining association rules related to timeout on the executions of the isolated modes, we found that importing *unittests*, defining “raise” and “while” statements increases the frequency of timeouts by at least 9.81, 2.37 and 1.63 times, respectively, as presented in Table 3.5.

The bloated docker environments failed much less due to *ImportError* and *ModuleNotFoundError* than the other environments, since these exceptions are related to missing dependencies. On the other hand, these environments failed much more due to *AttributeError*, *Key-*

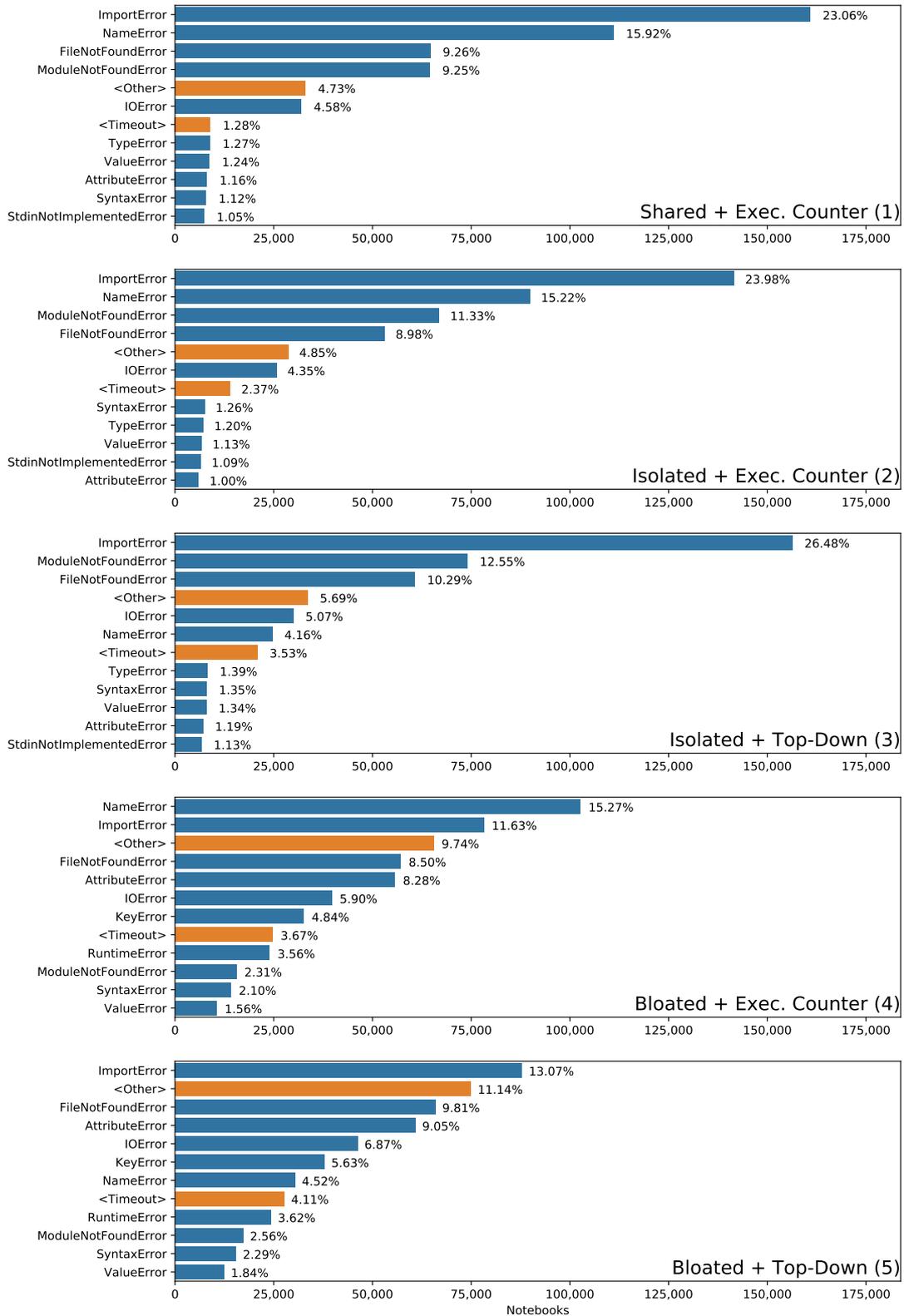


Figure 3.11: Failure reasons for the executions in each execution mode. The blue bars represent the Top 10 exceptions. The “Timeout” orange bar represents executions that we stopped when they took 5 minutes to run. The “Other” orange bar groups all the other exceptions that are not part of the Top 10.

Table 3.6: Association rules related to skips and *NameError*

Mode	Antecedent	Consequent	Support	Confidence	Lift
Isolated + Exec. Counter	Skips in the middle	NameError	13.41%	20.32%	1.39
Isolated + Exec. Counter	Skips	NameError	14.21%	18.50%	1.27
Isolated + Top-Down	Skips in the middle	NameError	3.05%	4.62%	1.18
Isolated + Top-Down	Skips	NameError	3.56%	4.63%	1.19

*Error*, and *RuntimeError*. The former two exceptions are related to updates on libraries that deprecate and change APIs, and the latter exception may be related to conflicting versions of libraries. Hence, simply installing dependencies from imports does not solve all dependency problems. In fact, with the growth of these other types of exceptions, the percentage of notebooks that run all cells did not improve much. These issues would be better addressed through the proper definition of dependencies with their versions in dependency files.

Surprisingly, in the first execution mode (in which we tried to install declared dependencies), 44.18% of the notebooks from repositories with declared dependencies failed with one of these errors. In contrast, only 31.61% of the notebooks from repositories without declared dependencies failed with these errors. It probably happened because we used anaconda environments with more pre-installed dependencies for the latter ones. Still, it indicates that many dependency files do not declare all the notebook dependencies. Popular notebooks were about 13% more likely to fail in both situations.

Another very common exception in the executions that followed the cell execution counter was *NameError*. This exception occurs when Python tries to access a variable that was not defined. This exception is related to hidden states and out-of-order cells. By mining association rules over features from executions of the isolated modes, we found that skips raise the chance of *NameError* exceptions, as presented in Table 3.6. Skips increase more the likelihood of exceptions when we run notebooks following the cell execution counter than when we run them in top-down order (1.27 times vs. 1.19 times). Moreover, having skips in the middle raises even more the likelihood of exception in the execution counter order (1.39 times). Since this exception occurred much more on executions that followed the execution counter than on executions that followed the top-down order, this suggests that even though users can execute the notebook at any other and define and redefine variables, they still tend to define variables in the top-down order.

Finally, the other very common exceptions were *FileNotFoundError* and *IOError*. These

errors occur when users use absolute paths to access data files or do not include the data in the repositories.

**Reproducibility Results.** Table 3.7 presents the reproducibility results for each execution mode, considering each normalization described in Table 3.2, and Table 3.8 presents the reproducibility results for the popular notebooks. The normalization columns of the first execution mode (shared + execution counter) are empty because we originally did not apply any normalization. In these tables, the percentages refer to the number of notebooks that we attempt to run (i.e., we exclude failed installations in the first execution mode and notebooks with dependency descriptors in isolated modes). Also, we considered notebooks that resulted in a timeout and notebooks with exceptions as non-reproducible. This last assumption may not be true since reproducible notebooks could have exceptions in them.

In Table 3.7, we calculated that about 11% of the executed notebooks (or 6% in Table 3.8 for the popular group) originally had exceptions, and about 7% of the notebooks (popular group: 4% in Table 3.8) had cells with outputs after the original exception. While we counted them as non-reproducible, these exceptions are likely the expected behavior of these notebooks. However, as we did not compare the exceptions, we cannot indicate whether they are reproducible.

The percentage of executions that run all cells ranged from 22.57% to 26.09%. These results are very close to the reproducibility rate of 24.9% that Collberg et al. (2014) achieved in their study of reproducibility in general computer systems research. Their study only attempted to compile the source code and did not check the execution results. In our case, the shared + execution counter mode was able to run more notebooks (26.09%) than the other modes, probably due to the installation of dependency files. Additionally, the bloated + execution counter mode had the smallest percentage of notebooks that run all cells (22.57%). However, these results do not reflect the number of notebooks that produce the same results.

The worst reproducibility rate was observed for the first execution mode (shared + execution counter) with no normalization (4.90%) and the best rate occurred for the bloated + top-down mode after the image normalization (15.04%), as expected. The image normalization not only applies all the previous normalizations shown in Table 3.2, but it is also the most susceptible to false positives, as it ignores differences in image results by considering that these differences could be caused by small changes in module updates. The boated + execution counter mode had the second-best results (14.40%, after the image normalization).

Popular notebooks were more reproducible in all situations. The percentage of executions that run all cells ranged from 31.84% to 36.27%. In this case, the isolated + top-down mode

Table 3.7: Reproducibility results for all notebooks.

	Shared + Exec. Counter (1)	Isolated + Exec. Counter (2)	Isolated + Top Down (3)	Bloated + Exec. Counter (4)	Bloated + Top Down (5)
Attempted executions	697,398	590,354	590,358	672,232	672,235
Run all cells	181,955 (26.09%)	137,208 (23.24%)	152,555 (25.84%)	151,730 (22.57%)	170,949 (25.43%)
Stopped by timeout	8,903 (1.28%)	13,969 (2.37%)	20,847 (3.53%)	24,690 (3.67%)	27,652 (4.11%)
Stopped by exception	506,539 (72.63%)	439,177 (74.39%)	416,956 (70.63%)	495,296 (73.68%)	473,324 (70.41%)
Had exception originally	80,520 (11.55%)	67,762 (11.48%)	66,226 (11.22%)	76,151 (11.33%)	74,816 (11.13%)
Output after exception	55,138 (7.91%)	46,165 (7.82%)	43,459 (7.36%)	52,042 (7.74%)	49,287 (7.33%)
<b>Same results</b>					
No normalization	34,148 (4.90%)	29,927 (5.07%)	33,555 (5.68%)	58,365 (8.68%)	58,910 (8.76%)
Encode		29,927 (5.07%)	33,555 (5.68%)	58,365 (8.68%)	58,910 (8.76%)
Execution counter		39,976 (6.77%)	44,618 (7.56%)	70,050 (10.42%)	71,873 (10.69%)
Stream		42,306 (7.17%)	47,274 (8.01%)	72,449 (10.78%)	74,410 (11.07%)
Dictionary		42,306 (7.17%)	47,426 (8.03%)	72,449 (10.78%)	74,410 (11.07%)
Dataframe		45,773 (7.75%)	51,357 (8.70%)	75,708 (11.26%)	77,990 (11.60%)
Exception path		45,773 (7.75%)	51,415 (8.71%)	75,708 (11.26%)	77,990 (11.60%)
Deprecation		48,138 (8.15%)	54,058 (9.16%)	79,763 (11.87%)	82,370 (12.25%)
White space		48,138 (8.15%)	54,593 (9.25%)	79,763 (11.87%)	82,370 (12.25%)
Decimal		48,138 (8.15%)	55,161 (9.34%)	79,763 (11.87%)	82,370 (12.25%)
Date		48,138 (8.15%)	55,174 (9.35%)	79,763 (11.87%)	82,370 (12.25%)
Time		48,138 (8.15%)	55,183 (9.35%)	79,763 (11.87%)	82,370 (12.25%)
Memory		48,138 (8.15%)	55,404 (9.38%)	79,763 (11.87%)	82,370 (12.25%)
Image		64,214 (10.88%)	76,745 (13.00%)	96,783 (14.40%)	101,078 (15.04%)

Table 3.8: Reproducibility results for the popular group.

	Shared + Exec. Counter (1)	Isolated + Exec. Counter (2)	Isolated + Top Down (3)	Bloated + Exec. Counter (4)	Bloated + Top Down (5)
Attempted executions	19,473	13,842	13,842	17,411	17,411
Run all cells	6,864 (35.25%)	4,806 (34.72%)	5,021 (36.27%)	5,543 (31.84%)	5,858 (33.65%)
Stopped by timeout	172 (0.88%)	280 (2.02%)	363 (2.62%)	639 (3.67%)	679 (3.90%)
Stopped by exception	12,437 (63.87%)	8,756 (63.26%)	8,458 (61.10%)	11,224 (64.46%)	10,869 (62.43%)
Had exception originally	1,201 (6.17%)	861 (6.22%)	843 (6.09%)	1,046 (6.01%)	1,032 (5.93%)
Output after exception	873 (4.48%)	616 (4.45%)	584 (4.22%)	758 (4.35%)	727 (4.18%)
<b>Same results</b>					
No normalization	2,135 (10.96%)	1,610 (11.63%)	1,652 (11.93%)	2,590 (14.88%)	2,609 (14.98%)
Encode		1,610 (11.63%)	1,652 (11.93%)	2,590 (14.88%)	2,609 (14.98%)
Execution counter		1,769 (12.78%)	1,821 (13.16%)	2,769 (15.90%)	2,801 (16.09%)
Stream		2,097 (15.15%)	2,142 (15.47%)	2,907 (16.70%)	2,935 (16.86%)
Dictionary		2,097 (15.15%)	2,146 (15.50%)	2,907 (16.70%)	2,935 (16.86%)
Dataframe		2,189 (15.81%)	2,243 (16.20%)	2,999 (17.22%)	3,029 (17.40%)
Exception path		2,189 (15.81%)	2,243 (16.20%)	2,999 (17.22%)	3,029 (17.40%)
Deprecation		2,249 (16.25%)	2,310 (16.69%)	3,098 (17.79%)	3,134 (18.00%)
White space		2,249 (16.25%)	2,332 (16.85%)	3,098 (17.79%)	3,134 (18.00%)
Decimal		2,249 (16.25%)	2,343 (16.93%)	3,098 (17.79%)	3,134 (18.00%)
Date		2,249 (16.25%)	2,343 (16.93%)	3,098 (17.79%)	3,134 (18.00%)
Time		2,249 (16.25%)	2,343 (16.93%)	3,098 (17.79%)	3,134 (18.00%)
Memory		2,249 (16.25%)	2,347 (16.96%)	3,098 (17.79%)	3,134 (18.00%)
Image		2,678 (19.35%)	2,846 (20.56%)	3,641 (20.91%)	3,712 (21.32%)

could run more notebooks than the other environments, proportionally (36.27%). However, in terms of producing the same results, the bloated + top-down mode dominated all the others, reaching a reproducibility rate of 21.32% after the image normalization.

While the normalizations did not affect the ability to run notebooks, they almost doubled the reproducibility rate (compared to the scenario when no normalization was applied). However, not all normalizations were equally effective. The most effective normalizations were image, execution counter, dataframe, deprecation, and stream. The deprecation normalization had more effect than the dataframe one on the environments with all dependencies. This is probably due to the fact that we installed the most recent version of packages in the bloated environments, increasing the chance of having deprecations in them.

Table 3.9: Association rules related to executions that generate the same results after the execution counter normalization.

Mode	Antecedent	Consequent	Support	Confidence	Lift
Isolated + Exec. Counter	9 or less cells (1st quartile)	Same Results	3.84%	14.47%	2.22
Isolated + Exec. Counter	while	Same Results	0.91%	11.95%	1.83
Isolated + Exec. Counter	class	Same Results	0.88%	10.94%	1.67
Isolated + Exec. Counter	Skips in the middle	Same Results	2.77%	4.19%	0.64
Isolated + Exec. Counter	Imports	Same Results	3.66%	4.11%	0.63
Isolated + Exec. Counter	Unordered	Same Results	0.80%	2.23%	0.34
Isolated + Exec. Counter	37 or more cells (4th quartile)	Same Results	0.44%	1.98%	0.30
Isolated + Top-Down	9 or less cells (1st quartile)	Same Results	4.00%	15.09%	2.07
Isolated + Top-Down	while	Same Results	1.00%	13.16%	1.81
Isolated + Top-Down	class	Same Results	0.96%	12.02%	1.65
Isolated + Top-Down	Skips in the middle	Same Results	3.48%	5.28%	0.73
Isolated + Top-Down	Imports	Same Results	4.29%	4.81%	0.66
Isolated + Top-Down	Unordered	Same Results	1.55%	4.31%	0.59
Isolated + Top-Down	37 or more cells (4th quartile)	Same Results	0.59%	2.62%	0.36

Since the execution counter normalization is among the ones that affected the most the reproducibility rates without adding false positives, we decided to use it when mining association rules. The association rules indicate that small notebooks, “while” definitions, and “class” definitions raise the probability of obtaining the same results by 122%, 83%, and 67% when

following the execution counter order, and 107%, 81%, and 65% when following the top-down order, respectively. On the other hand, we found that skips in the middle, imports, unordered cells, and big notebooks decrease the chance of obtaining the same results by 36%, 37%, 66%, and 70% when following the execution counter order, and 27%, 34%, 41%, and 64% when following the top-down order, respectively, as presented in Table 3.9.

**RQ.N7.** *How reproducible are notebooks?*

**Answer:** We were able to successfully run between 22.57% and 26.09% of the notebooks that we attempted to run. This number is close to the results of a previous reproducibility study (COLLBERG et al., 2014) about general computer systems research (24.9%). However, the rates are way smaller (4.90% – 15.04%) when we count only notebooks that produce the same results. The most common causes of failures were related to missing dependencies, the presence of hidden states and out-of-order executions, and data accessibility in all execution sets. In the experiments that we used docker environments with most pre-installed dependencies, many executions also failed due to incompatible versions of dependencies and conflicts.

**Implications:** While the reproducibility rate is comparable to the rate in general computer systems research (COLLBERG et al., 2014), it is far from ideal. The identification of the root causes suggests that there is an opportunity to improve the reproducibility rate in notebooks by devising approaches that address these problems. More specifically, managing the dependencies of notebooks and guaranteeing the linear (top-down) execution order could improve the reproducibility rate. It is worthy of noting that dependency resolution problems are also common in other contexts, such as building past snapshots of software (TUFANO et al., 2017). Additionally, tools such as ReproZip (CHIRIGATI et al., 2016) can automatically capture dependencies (both libraries and data) and create packages including these dependencies, thus ensuring reproducibility. ReproZip has a plugin for Jupyter (REPROZIP, 2017).

## 3.5 Threats to Validity

This study attempts to obtain a picture of quality and reproducibility practices used in the design of Jupyter Notebooks. As presented in Section 3.3, we have designed measures that capture different aspects of notebooks that impact their reproducibility. These measures, however, have some threats to validity that we discuss below.

**Internal.** While we used clean conda environments in the first execution mode (shared + execution counter), we did not isolate the executions in the system. It means that a notebook execution or dependency installation could install or modify system dependencies before the preparation and execution of another notebook. We attempted to minimize this threat by running more analyses in isolated docker environments. However, in the additional analyses, we did not attempt to install the dependencies declared in the repositories, due to time constraints. Instead, we did try to install all modules imported by the notebooks in separate environments.

Additionally, we examined all notebooks from GitHub as valid subjects in this chapter. We did not account for all the perils of mining repositories (KALLIAMVAKOU et al., 2014). Some analyzed notebooks may not be intended to be reproducible and may not value quality. For instance, students prepare exercises with the goal of studying for a course. These exercises have a short life-span and are often not classified as engineered software projects (MUNIAH et al., 2017). A basic check for notebooks containing words related to exercises (“assignment”, “course”, “exercise”, “homework”, “lesson”) returns 164,463 unique notebooks (16.06%). Even though this check is very susceptible to false positives and false negatives, it indicates that exercises are a solid use case for notebooks and deserve investigations. Other use cases for notebooks (e.g., tutorial notebooks, research notebooks, dashboards, and others) may also have different goals in terms of quality and reproducibility and also require further investigations. In the sampled notebooks, we observed that education is a big use case for notebooks. Hence, even though these notebooks may have different goals in terms of quality and reproducibility, it is still worth it to understand them to improve the support for these aspects.

Moreover, during sampling, we manually analyzed the characteristics of the notebooks. This analysis is subject to human error. We attempted to mitigate this threat by comparing some results with proxies on the database. However, these proxies are not complete (i.e., there are things that we only observed in the sample) nor reliable for qualitative analyses (i.e., they do not capture nuances that we could interpret by reading the notebook).

**Construct.** The methods we use to answer the research questions aim to attain an approximated answer since it is not possible to get accurate answers that precisely represent all notebooks without false positives and false negatives. For instance, a module for statistical tests could have “test” in its name and appear as an answer to **RQ.N4** without being a module for testing software. Similarly, we may not detect a testing module that does not have “test” or “mock” in its name, and that does not appear in the testing tools taxonomy (PYTHON-WIKI, 2019).

Moreover, in the reproducibility study, we did not consider the maintainability of notebooks

and libraries. Many libraries might have been updated since the notebooks were originally developed. This should be a threat for the bloated execution modes, which uses arbitrary versions of the libraries. However, we found that these modes were more reproducible than the shared environment, which attempted to install pinned versions declared in dependency files (`setup.py`, `requirements.txt`, and `Pipfile`). Similarly, many repositories may have been updated to account for library changes since we first collected them. However, when assessing the maintainability of repositories with notebooks, we found that only 12.68% of them still had some active development six months after the collected commit, and only 2.84% of them were still active in the six months prior to the moment we queried GitHub again (July 22nd, 2020). Hence, it is reasonable to assume that most repositories are not maintained and perform the reproducibility study as is.

Additionally, we only checked whether the notebooks generated the same results when they successfully ran all cells. However, we stopped the executions on exceptions and did not consider these notebooks as reproducible. An exception may be the expected (although unusual) behavior of a notebook, and it may have executed code after the exception as well.

To account for small deviations in the notebooks' results that were leading to false negatives in the analyses of same results, we performed normalizations on the outputs. While some normalizations reduce the number of false negatives without drawbacks (e.g., encode normalization and execution counter normalization), other normalizations increase false positives. For instance, after applying the image normalization, two notebooks can generate completely different images, but we will consider them as generating the same results. To assess the popularity of notebooks, we used the number of stars and forks from repositories as a proxy for the notebooks due to the lack of a better number. Since these numbers are from the repositories, they may not represent the popularity of a notebook. For instance, a tool repository that uses a notebook as an example of how to use it may be popular because of the tool and not because of the notebook. However, in our comparisons, popular notebooks had more quality features and reproducibility than the overall group, indicating that the proxy was sound.

**External.** We collected repositories from GitHub for over one year. During this period, many repositories were updated, and many repositories were removed. Despite having data until April 16th, 2018, the repository states represent their state during the collection and not their state on this date. Additionally, we restricted our analysis to committed notebooks. Presumably, these notebooks receive more attention than the average scratchpad notebook and follow better practices. For instance, Grus (2018) pointed out the problem of untitled notebooks, but in our data, these notebooks correspond only to 1.93% of the notebooks.

## 3.6 Discussion

In this chapter, we analyzed evidence of good and bad practices on the development of Jupyter Notebooks regarding quality and reproducibility by going through the main criticisms that the format receives (GRUS, 2018; MUELLER, 2018; POMOGAJKO, 2015). In our experimental results, which we discuss part in this chapter and part in the Mining Software Repositories paper (PIMENTEL et al., 2019b), we found evidence of both good and bad practices. As good practices, we found the usage of literate programming aspects of notebooks (e.g., markdown cells and visualizations), the application of abstractions on notebooks that have more complex control flows, and the usage of descriptive filenames. As bad practices, we found that most notebooks do not test their code and that a large number of notebooks has characteristics that hinder the reasoning and the reproducibility, such as out-of-order cells, non-executed code cells, and the possibility of hidden states.

In comparison to the scripts analyzed in Chapter 2, Python scripts seem to use more Loop (96.51% of scripts) and Condition (93.02%) constructs than interactive notebooks with Python code (47.48% and 63.30%, respectively). Other constructs appear at a similar frequency with a margin of 15% of difference. Regarding the most used modules, four modules appear in the top 10 of both studies: `numpy`, `pandas`, `os`, and `math`, but at a different order. Python scripts seem to use more built-in modules (seven of the top 10), while notebooks seem to use more external ones (once again, seven of the top 10).

Despite these small differences, notebooks could also benefit from provenance from scripts to understand the execution order and to minimize the effects of hidden states. Additionally, notebooks lack guidelines and linting tools to minimize these problems.

# Chapter 4

## State-of-the-Art on Provenance from Scripts

### 4.1 Introduction

Computing has 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. Systematic mechanisms to collect provenance for computational experiments are critical to address these challenges.

As previously discussed, compared to SWfMS (CALLAHAN et al., 2006; LIN et al., 2009; WOLSTENCROFT et al., 2013; ZHAO et al., 2007; FREIRE et al., 2006; OLIVEIRA et al., 2010), 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 (MWEBAZE; BOXHOORN; VALENTIJN, 2009; BOCHNER; GUDE; SCHREIBER, 2008; MURTA et al., 2014; LERNER; BOOSE, 2014b; DAVISON, 2012; MCPHILLIPS et al., 2015b). Each one of these approaches proposes different mechanisms for collecting, managing, and analyzing different types of provenance in scripts with multiple goals. In this chapter, we propose a classification taxonomy for approaches that work with provenance from scripts.

Multiple surveys have been written about provenance. Some characterize data provenance in e-Science (SIMMHAN; PLALE; GANNON, 2005a; GLAVIC; DITTRICH, 2007), provenance in computational tasks in general (FREIRE et al., 2008), provenance in databases (TAN et al., 2007), data-intensive scientific workflow management (LIU et al., 2015), and provenance in the light of Big Data (WANG, Jianwu et al., 2015). Others focus on more specific aspects,

such as dynamic steering (MATTOSO et al., 2015) and provenance analytics (OLIVEIRA; OLIVEIRA; BRAGANHOLO, 2018). Finally, Herschel, Diestelkämper, and Lahmar (2017) characterizes provenance in general, considering use cases, types of provenance and system requirements. However, none of these surveys consider the specific use of provenance from scripts. In this chapter, we aim to fill this gap by providing a comprehensive survey of existing techniques that address scripts-related problems.

For preparing the comprehensive list of techniques we discuss in this chapter, we first conducted a systematic mapping (PETERSEN et al., 2008) to identify the state-of-the-art tools on provenance from scripts, as discussed in Section 4.2. Then, we observed how each one of these approaches collects, manages, and analyzes provenance from scripts and organized the techniques into the taxonomy proposed in Section 4.3. 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. In this thesis, we use the taxonomy in Chapter 5 to describe our approach for collecting provenance from scripts. Section 4.4 presents the threats to validity of the snowballing process and the taxonomy proposal.

This chapter was published in the ACM Computing Surveys with a more detailed classification of the approaches according to the taxonomy (PIMENTEL et al., 2019a). We opted not to include the categorization here for conciseness. Section 4.5 presents a systematic update on the list of state-of-the-art tools. Finally, Section 4.6 discusses the obtained results.

## 4.2 Related Work

For constructing the list of existing techniques, we conducted a systematic mapping (PETERSEN et al., 2008) to identify the state-of-the-art tools on provenance from scripts. According to Petersen et al. (2008), 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.

We applied forward and backward snowballing to discover relevant tools (WOHLIN, 2014). 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 (GROTH; MILES; MOREAU, 2005)), generic binary executables (e.g., ReproZip (CHIRIGATI; SHASHA; FREIRE, 2013), DataTracker (STAMATOIANNAKIS; GROTH; BOS, 2014)), or approaches that collect provenance at the operating system (OS) layer (e.g., PASS (MUNISWAMY-REDDY et al., 2006), Burrito (GUO; SELTZER, 2012)). 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 also excluded approaches that use scripts for defining workflows but restrict them to constructs that support the creation of a DAG, such as Swift (ZHAO et al., 2007), Snake-make (KÖSTER; RAHMANN, 2012), dispel4py (FILGUIERA et al., 2014), signac (RAMASUBRAMANI et al., 2018), PRUNE (IVIE, 2018), W2Share (CARVALHO; BELHAJJAME; MEDEIROS, 2018), and SoS (WANG; PENG, 2019). While such tools reduce the learning curve for scientists who are used to programming languages, they lack the flexibility provided by scripts. Hence, we considered them SWfMS. Similarly, we excluded Vizier (BRACHMANN et al., 2019; BRACHMANN; SPOTH, 2020), an approach that defines a restricted notebook system with isolated cells that represent blocks in an SWfMS (VisTrails).

We followed the guidelines proposed by Wohlin (2014) for defining the start set of our snowballing (i.e., use Google Scholar to avoid bias towards 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 (DEY et al., 2015; FREW; METZGER; SLAUGHTER, 2008; FREW; SLAUGHTER, 2008; HUQ; APERS; WOMBACHER, 2013b; LERNER; BOOSE, 2014b; MACKO; SELTZER, 2012; MCPHILLIPS et al., 2015a,b; MURTA et al., 2014) related to seven approaches, and we stopped on page 5 after the page did not contribute with 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 tool<sup>1</sup> until no more related papers were obtained. We finished this process on March 6th, 2017. Figure 4.1 presents the process and the amount of related and found papers in

---

<sup>1</sup><https://joaofelipe.github.io/snowballing/>

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 24th, 2016 found only 24 papers that cited the first noWorkflow paper (MURTA et al., 2014), 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 4.1, we present only what it would be if we had performed the whole snowballing on March 6th, 2017, with big backward and forward iterations, as described by Wohlin (2014). 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. Figure 4.2 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/>.

Table 4.1 presents the final selection of approaches with their papers. In this table, we categorized the approaches by their usage goals for provenance to understand the purpose of provenance in these tools. 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 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. 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. We also identified the main usage goal described in the papers. In this case, the order is comprehension, management, and reproducibility. Colors in Figure 4.2 represent the main usage goals of the approaches.

We grouped papers according to their publishing place to understand where these tools were published. We identified 42 papers published in conferences, 14 articles in journals, and 5

theses. Figure 4.3 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. 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. These results indicate that these venues are interested in the topic and that the topic is attracting attention from the international community.

In the following section, we propose a taxonomy for provenance from scripts based on the techniques proposed and applied by each of these approaches. After the proposal of the taxonomy, we updated the list of related work as we present in Section 4.5.



Table 4.1: Selected approaches with provenance support: main and secondary goals. Labels in secondary goals column refer to goals: *Cache* – Caching; *Compr* – Comprehension; *Frame*—Framework; *Manag* – Management; *Repro* – Reproducibility.

Approach	Main goal	Secondary goals				
		Cache	Compr	Frame	Manag	Repro
Astro-WISE (MWEBAZE; BOXHOORN; VALENTIJN, 2009, 2011)	Framework	✓	✓	✓	✗	✓
Becker and Chambers (1988)	Comprehension	✗	✓	✗	✗	✓
Bochner, Gude, and Schreiber (2008)	Comprehension	✗	✓	✓	✗	✗
CPL (MACKO; SELTZER, 2012)	Framework	✗	✗	✓	✗	✗
CXXR (SILLES; RUNNALLS, 2010; RUNNALLS, A.; SILLES, C., 2012; SILLES, 2014; RUNNALLS, A. R.; SILLES, C. A., 2011; RUNNALLS, 2011)	Comprehension	✗	✓	✗	✗	✗
Datatrack (EICHINSKI; ROE, 2016)	Management	✗	✓	✗	✓	✗
ES3 (FREW; METZGER; SLAUGHTER, 2008; FREW, 2004; FREW; SLAUGHTER, 2008; VALEUR, 2005; FREW; JANÉE; SLAUGHTER, 2010, 2011)	Comprehension	✗	✓	✗	✗	✗
ESSW (FREW; BOSE, 2001)	Management	✗	✓	✗	✓	✗
IncPy (GUO; ENGLER, D. R., 2010; GUO; ENGLER, D., 2011; GUO, 2012)	Caching	✓	✗	✗	✗	✗
Lancet (STEVENS; ELVER; BEDNAR, 2013)	Reproducibility	✗	✓	✗	✓	✓
Magni (OXVIG; ARILDSEN; LARSEN, 2016)	Reproducibility	✗	✗	✓	✗	✓
Michaelides et al. (2016)	Reproducibility	✗	✓	✗	✗	✓
noWorkflow (MURTA et al., 2014; PIMENTEL et al., 2015, 2016b,c)	Comprehension	✗	✓	✗	✓	✓

Continued on next page

Approach	Main goal	Secondary goals				
		Cache	Compr	Frame	Manag	Repro
Provenance Curious (HUQ; APERS; WOMBACHER, 2013a,b; HUQ, 2013)	Comprehension	✗	✓	✗	✗	✗
pypet (MEYER; OBERMAYER, 2015, 2016)	Management	✗	✗	✗	✓	✗
RDataTracker (LERNER; BOOSE, 2014a,b)	Comprehension	✗	✓	✗	✗	✗
Sacred (GREFF; SCHMIDHUBER, 2015)	Management	✗	✓	✗	✓	✗
RFlow (CRUZ; NASCIMENTO, 2016)	Management	✗	✓	✗	✓	✓
SPADE (TARIQ; ALI; GEHANI, 2012)	Comprehension	✗	✓	✗	✗	✓
StarFlow (ANGELINO; YAMINS; SELTZER, 2010; ANGELINO et al., 2011)	Management	✓	✓	✗	✓	✓
Sumatra (DAVISON, 2012)	Reproducibility	✗	✓	✗	✓	✓
Variolite (KERY; HORVATH; MYERS, 2017)	Management	✗	✓	✗	✓	✓
VCR (GAVISH; DONOHO, 2011)	Reproducibility	✗	✓	✗	✗	✓
versuchung (DIETRICH; LOHMANN, 2015)	Reproducibility	✗	✓	✓	✗	✓
WISE (ACUÑA; LACROIX; BAZZI, 2015; ACUÑA, 2015; ACUÑA; CHOMILIER; LACROIX, 2015; ACUÑA; LACROIX, 2016)	Comprehension	✗	✓	✗	✗	✗
YesWorkflow (MCPHILLIPS et al., 2015a,b)	Comprehension	✗	✓	✗	✗	✗
YW*NW (DEY et al., 2015; PIMENTEL et al., 2016a)	Comprehension	✗	✓	✗	✓	✓
<b>Main Goal / Total</b>		1 / 3	11 / 23	2 / 5	7 / 11	6 / 14

### 4.3 Taxonomy

Many approaches have been proposed to collect provenance from binary executions (e.g., Re-proZip (CHIRIGATI; SHASHA; FREIRE, 2013), PASS (MUNISWAMY-REDDY et al., 2006),

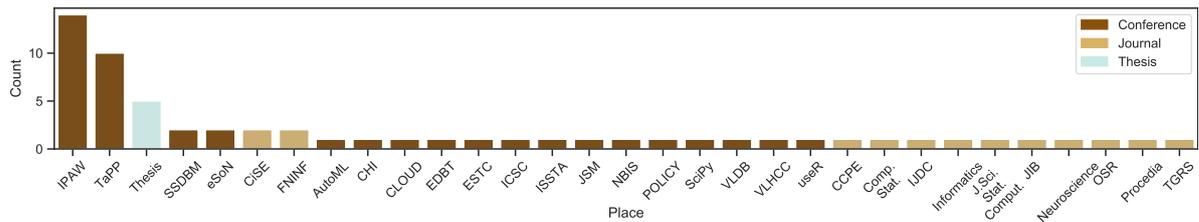


Figure 4.3: Distribution of work by publishing location.

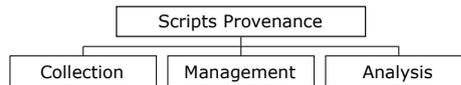


Figure 4.4: Main taxonomy of provenance from scripts.

CDE (GUO; ENGLER, D. R., 2011), DataTracker (STAMATOIANNAKIS; GROTH; BOS, 2014), 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 share 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 the output to the input nor help users to understand the experiments. On the other hand, the literate programming paradigm (KNUTH, 1984) 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, as we discuss in Chapter 3. Some tools that use scripting languages and support literate programming, such as Jupyter (SHEN et al., 2014), may also benefit from additional provenance collected from scripts (PIMENTEL et al., 2015).

```

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))

```

Figure 4.5: Toy experiment that classifies a yearly precipitation data from Rio de Janeiro.

### 4.3.1 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 4.5 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.

Cheney, Chiticariu, and Tan (2007) 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 4.5 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 4.5, 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 4.5 identifies that “p13.dat” was obtained from BDMEP<sup>2</sup> 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 4.5 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 file locations as where-provenance. However,

<sup>2</sup>BDMEP is a meteorological database for teaching and research.

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* (LIM et al., 2010; ZHAO; WILDE; FOSTER, 2006). Retrospective provenance combines why-provenance and how-provenance to provide an understanding of the execution process, identifying what really happened during the execution. On the other hand, 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 4.5 includes the script itself and the modules “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 chapter, this classification system encodes too much information in the prospective provenance, and lack a different type of provenance.

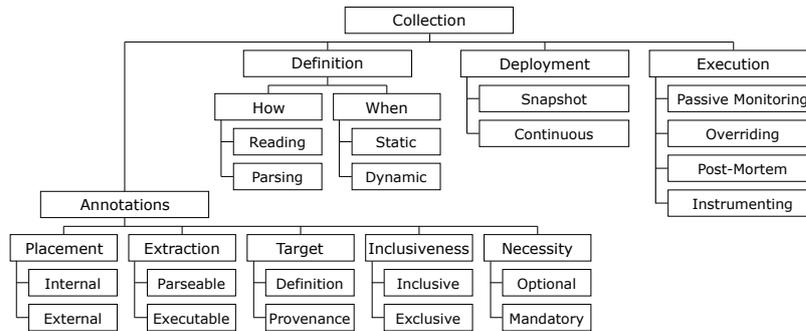
Clifford et al. (2008) propose a similar classification with three categories: *program structure*, *runtime logs*, and *annotations*. In this system, runtime logs correspond to retrospective provenance and program structure corresponds to the structural part of the prospective provenance. This system does not consider environment information. The third category in this system, annotations, refer to user-made annotations in the provenance or structure, which allow users to explain the program. In Figure 4.5, 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. (2014) borrows terms from software engineering (VAN DER HOEK, 2004) 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. (2008). In Figure 4.5, 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 4.5, 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 4.6 presents the aforementioned classification systems for provenance. Note that

Cheney <i>et al.</i> (2007)			For databases → Why	How	Where
Zhao <i>et al.</i> (2006)		Prospective		Retrospective	
Clifford <i>et al.</i> (2008)	Annotations	Structure		Runtime Logs	
Murta <i>et al.</i> (2014)		Definition	Deployment	Execution	
<b>This document</b>	Annotations	Definition	Deployment	Execution	

Figure 4.6: Provenance classification systems.

Figure 4.7: Expanded *Collection* taxonomy node of Figure 4.4.

for the proposed taxonomy, we use the classification proposed by Murta *et al.* (2014) due to its explicit separation of definition and deployment provenance, together with the *annotations* provenance proposed by Clifford *et al.* (2008).

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 (ANGELINO; YAMINS; SELTZER, 2010; MCPHILLIPS *et al.*, 2015a). In this section, we present different collection mechanisms for each provenance type. Figure 4.7 presents the collection taxonomy.

#### 4.3.1.1 Annotations

According to Clifford *et al.* (2008), users can make annotations either on procedures or on data. Additionally, we identify that some approaches also support annotations on provenance itself (DAVISON, 2012). 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 (DAVISON, 2012). Additionally, annotations can facilitate collecting other provenance types (ANGELINO; YAMINS; SELTZER, 2010; LERNER; BOOSE, 2014b; MCPHILLIPS *et al.*, 2015a). We classify annotations in five axes, as presented

in Figure 4.7: placement, extraction, target, inclusiveness, and necessity.

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 4.5. On the other hand, 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 *executable* or *parseable*. Provenance systems can extract parseable annotations statically. However, executable annotations require their execution. In Figure 4.5, “where” in line 4 is an executable annotation, as it is necessary to execute it to get its result. However, the commentaries on lines 10-11 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 4.5 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 4.5 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, the annotations are mandatory. Otherwise, if it only uses annotations to enrich or filter the provenance collection, annotations are optional.

#### 4.3.1.2 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 (CRUZ; NASCIMENTO, 2016; DAVISON, 2012) or as fine-grained as extracting structure information from scripts to describe them (MCPHILLIPS et al., 2015b; HUQ; APERS; WOMBACHER, 2013b; MURTA et al., 2014).

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 (ESTUBLIER, 2000)

can help with definition provenance collection (DAVISON, 2012). 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 4.3.1.4 and collecting the files as soon as the execution tries to access it (CRUZ; NASCIMENTO, 2016; MURTA et al., 2014). This is especially valid for scripts since interpreters read their definitions before running them. However, this strategy may generate only a partial definition provenance of the project according to the execution path (GUO; ENGLER, D. R., 2011).

For finer-grained collection, it is necessary to statically analyze the structure (HUQ; APERS; WOMBACHER, 2013b). Due to the unpredictability of dynamic languages (XU et al., 2013), 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 (ANGELINO; YAMINS; SELTZER, 2010; MCPHILLIPS et al., 2015b). 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 4.7. Definition provenance can be collected by *reading* whole files or *parsing* files and extracting information from them. In Figure 4.5, if we collect the whole script file, we will have definition provenance by reading. On the other hand, if we parse the file and extract information from it, 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 4.5, 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.

### 4.3.1.3 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. It is the case for module and program dependencies and environment variables. Hence, the strategies we describe in Section 4.3.1.4 for execution

provenance also apply for continuously collecting such deployment provenance during execution (CHIRIGATI; SHASHA; FREIRE, 2013). 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 (DAVISON, 2012; MURTA et al., 2014) to avoid the overhead of dynamic provenance collection (CHENEY; AHMED; ACAR, 2011).

As presented in Figure 4.7, we classify deployment provenance according to its collection frequency, as *snapshot* or *continuous*. In Figure 4.5, 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.

#### 4.3.1.4 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. On the other hand, 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 (KOOP et al., 2010), some collection mechanisms presented in this section support collecting not only meta-data 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, Metzger, and Slaughter (2008), 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 code explicitly with annotations or function calls. We identify a fourth strategy: post-mortem, which infers execution provenance after the execution (DAVISON, 2012; HUQ; APERS; WOMBACHER, 2013a; MCPHILLIPS et al., 2015a).

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, on the other hand,



Figure 4.8: Observed and disclosed strategies.

require users to specify what they want to collect, being error-prone and producing less provenance. Braun et al. (2006) 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 4.8 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 (GUO; ENGLER, D. R., 2011) or to use language-specific tracers to collect provenance (ANGELINO; YAMINS; SELTZER, 2010; MURTA et al., 2014). This chapter focuses on the latter. In Figure 4.5, 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 4.5, 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). It occurs because

dynamic program slicing and dynamic taint tracking just observe what occurred and not what could occur in other conditions (GUO; ENGLER, D. R., 2011). 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. In order 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 (DAVISON, 2012). Alternatively, it is possible 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 (HUQ; APERS; WOMBACHER, 2013a). Another way to apply the post-mortem strategy is to use annotations (MCPHILLIPS et al., 2015a) 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 (ACUÑA, 2015). In Figure 4.5, 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 (MARINHO et al., 2011)). In Figure 4.5, 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 “classification.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 (ANGELINO; YAMINS; SELTZER, 2010) or invoke library functions (BOCHNER; GUDE; SCHREIBER, 2008; FREW; BOSE, 2001). 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 (ANGELINO; YAMINS; SELTZER, 2010; MCPHILLIPS et al., 2015b). For this reason, PrIME (MILES et al., 2011) 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 prove-

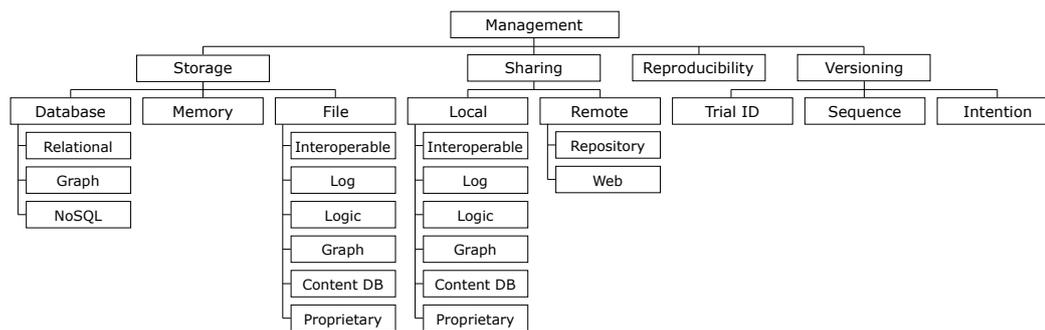


Figure 4.9: Expanded *Management* taxonomy node of Figure 4.4.

nance and how to enrich the collected provenance (LERNER; BOOSE, 2014b). In Figure 4.5, the “where” call in line 4 is an application of the instrumentation strategy.

### 4.3.2 Provenance Management

Collecting provenance data is not enough for provenance tools. It is desirable to provide management features related to *storage*, *sharing*, *versioning*, and *reproducibility*. In this section, we present provenance management requirements and approaches. Figure 4.9 presents the management taxonomy.

#### 4.3.2.1 Storage

Provenance can be stored in *database* systems, transient *memory*, or *files*. However, 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 (DAVISON, 2012). On the other hand, 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 (MURTA et al., 2014; HUQ; APERS; WOMBACHER, 2013b). Although file systems are also viable for such non-file data, they require the provenance tools to implement their own serialization mechanisms (GUO; ENGLER, D., 2011; STEVENS; ELVER; BEDNAR, 2013; OXVIG; ARILDSEN; LARSEN, 2016).

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 file-

name, with an external index to relate the original file name and version to its hash (DAVISON, 2012; GUO; ENGLER, D., 2011; MURTA et al., 2014). 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 (MURTA et al., 2014). Such collision avoidance approaches are not necessary, should the collection keep only the most recent versions (MICHAELIDES et al., 2016).

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 (DAVISON, 2012; HUQ; APERS; WOMBACHER, 2013b; MURTA et al., 2014). 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 (MURTA et al., 2014). This necessity of transitive closures also motivated some tools to use *graph* databases and other *NoSQL* databases right away (BOCHNER; GUDE; SCHREIBER, 2008; CHEBOTKO et al., 2013; FREW; METZGER; SLAUGHTER, 2008; GREFF; SCHMIDHUBER, 2015; MACKO; SELTZER, 2012).

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 (DAVISON, 2012; MURTA et al., 2014).

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 (ANGELINO; YAMINS; SELTZER, 2010; FREW; METZGER; SLAUGHTER, 2008; LERNER; BOOSE, 2014b; MCPHILLIPS et al., 2015b; MICHAELIDES et al., 2016; STEVENS; ELVER; BEDNAR, 2013). Other tools (or the same tool) might open these documents for analysis (LERNER; BOOSE, 2014b; STAMATOIANNAKIS; GROTH; BOS, 2014; MCPHILLIPS et al., 2015b) or reproducibility (STEVENS; ELVER; BEDNAR, 2013). Additionally, provenance might not be stored at all, should the application consume it at runtime (SILLES; RUNNALLS, 2010). 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 share it through remote network connections and expect other applications to deal with the storage (TARIQ; ALI; GEHANI, 2012; BOCHNER; GUDE; SCHREIBER, 2008).

### 4.3.2.2 Sharing

Besides storing provenance data, another provenance management issue is on sharing provenance to other people or systems for analysis and reproducibility. Sharing provenance for analysis allows tools to implement standalone collection mechanisms (TARIQ; ALI; GEHANI, 2012) and transfer the analysis responsibility to specialized tools. Sharing provenance for reproducibility reduces the burdens of making computation experiments reproducible across platforms (CHIRIGATI; SHASHA; FREIRE, 2013).

Provenance tools that store provenance at a small set of *files* (ANGELINO; YAMINS; SELTZER, 2010; MCPHILLIPS et al., 2015b; FREW; METZGER; SLAUGHTER, 2008; LERNER; BOOSE, 2014b; MICHAELIDES et al., 2016; STEVENS; ELVER; BEDNAR, 2013) support sharing by simply sending the files to someone else. Other tools need to process provenance data and produce the desirable file format (MURTA et al., 2014). However, the desirable file format depends on its application. Logic programming formats (e.g., Prolog and Datalog files) support running queries with transitive closures (MCPHILLIPS et al., 2015b; MURTA et al., 2014). Graph formats (e.g., GraphViz files) allow visual analysis (ACUÑA, 2015; PIMENTEL et al., 2016a). 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 (MICHAELIDES et al., 2016). Finally, it is also possible to share provenance as executable logs (MICHAELIDES et al., 2016), 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 (MOREAU et al., 2011). The OPM specification heavily influenced the W3C PROV standard (MOREAU; MISSIER, 2012). 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 (COSTA et al., 2013; MISSIER et al., 2013a).

All these formats provide sharable provenance but do not deal with the problem of provenance transferring. Thus, we define them as *local* sharing. RDFa (ADIDA et al., 2008) 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 sharable provenance can be embedded. In order 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 (BOCHNER; GUDE; SCHREIBER, 2008; GROTH; MILES; MOREAU, 2005) and as *repositories* designed to share provenance and experiment definitions, encouraging the reuse of experiments of other people (JONES et al., 2016). Version control system repositories (ESTUBLIER, 2000) play a similar role in sharing experiments. However, they usually only share script definitions, and they make it hard to search for other types of provenance. On the other hand, such systems provide versioning for the experiments.

#### 4.3.2.3 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 from validating research findings and expanding their horizons with new data (BAGGERLY; COOMBES, 2009). 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 (DONOHO et al., 2009). This situation leads to a credibility crisis (IOANNIDIS, 2005).

In response to the credibility crisis, scientists proposed sharing not only findings but also data, programs, and environments (CLAERBOUT; KARRENBACH, 1992), making data as transparent and available as possible (HANSON; SUGDEN; ALBERTS, 2011). 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 (DAVISON, 2012; GUO; SELTZER, 2012).

According to Drummond (2009), 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 towards 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.

#### 4.3.2.4 Versioning

Many experiment results motivate repetitions in their life cycle (MATTOSO et al., 2010). 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 a trial 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 4.3.1 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, in order 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 (MURTA et al., 2014; STEVENS; ELVER; BEDNAR, 2013) 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 (DAVISON, 2012; PIMENTEL et al., 2016b).

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* (MURTA et al., 2014), or as complete as indicating the *evolution intention* (PIMENTEL et al., 2016b).

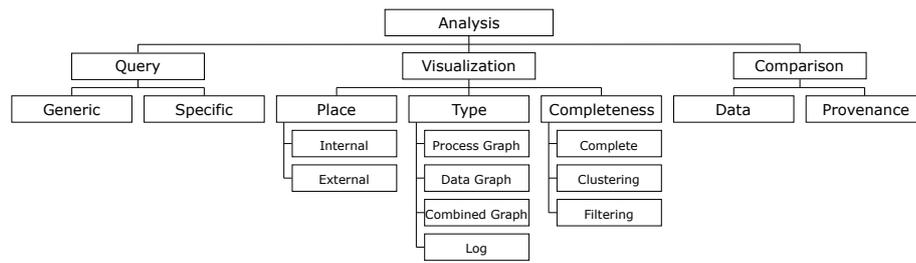


Figure 4.10: Expanded *Analysis* taxonomy node of Figure 4.4.

While provenance evolution has been applied to SWfMS (CALLAHAN et al., 2006), 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 (ESTUBLIER, 2000), 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 (DAVISON, 2012; STEVENS; ELVER; BEDNAR, 2013).

### 4.3.3 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 4.10 presents the analysis taxonomy.

#### 4.3.3.1 Query

Many approaches use *generic* languages for querying provenance, such as SQL (DAVISON, 2012; MURTA et al., 2014), SPARQL (MACKO; SELTZER, 2012; CHEBOTKO et al., 2010), XQuery (BOCHNER; GUDE; SCHREIBER, 2008), Prolog (MURTA et al., 2014), and Datalog (MCPHILLIPS et al., 2015b; ZHAO; LU, 2008). 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 (RAMAKRISHNAN; ULLMAN, 1995). 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 (MURTA et al., 2014). On the other hand, logic programming languages intuitively handle recursion.

Generic query languages are useful to users who know their syntax but can be complicated to deal with structured provenance data (FREIRE et al., 2008). 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 (LIM et al., 2013), VQuel (CHAVAN et al., 2015), and other proprietary ones for specific systems (LERNER; BOOSE, 2014b).

OPQL (LIM et al., 2013) 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 (CHAVAN et al., 2015) was proposed as a generalization of the Quel (STONEBRAKER et al., 1976) language with features of GEM (ZANIOLO, 1983) 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 (MATTOSO et al., 2015; SOUZA; MATTOSO, 2018). 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 (COSTA et al., 2013). 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 (GUO; ENGLER, D., 2011).

#### 4.3.3.2 Visualization

As we mentioned before, some approaches export provenance as interoperable files (e.g., OPM, PROV) for visualization in *external* tools (MICHAELIDES et al., 2016; TARIQ; ALI; GEHANI, 2012). 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 (ACUÑA, 2015; EICHINSKI; ROE, 2016; HUQ; APERS; WOMBACHER, 2013b; KOHWALTER et al., 2016; LERNER; BOOSE, 2014b; MCPHILLIPS et al., 2015b; MURTA et al., 2014).

Most approaches visualize provenance either as *logs* (GREFF; SCHMIDHUBER, 2015) or as directed *graphs* (ACUÑA, 2015; EICHINSKI; ROE, 2016; HUQ; APERS; WOMBACHER, 2013b; KOHWALTER et al., 2016; MCPHILLIPS et al., 2015b; LERNER; BOOSE, 2014b; MURTA et al., 2014). 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 (MCPHILLIPS et al., 2015b). *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 (MOREAU et al., 2011).

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 (EICHINSKI; ROE, 2016; HUQ; APERS; WOMBACHER, 2013b) or automatically (KOHWALTER et al., 2016; MURTA et al., 2014). 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 the sequence of provenance nodes (KOHWALTER et al., 2016) or just the information of a single node (MURTA et al., 2014). 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 (MURTA et al., 2014). Dynamic visualization tools can represent clusters as collapsible nodes (LERNER; BOOSE, 2014b).

It is possible to use query languages described in Section 4.3.3.1 for provenance filtering. Some query languages are distributed with provenance browsers that support provenance visualization (ANAND; BOWERS; LUDÄSCHER, 2010). 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 (KOHWALTER et al., 2016).

Graphs are not the only way to visualize provenance. Sankey Diagrams are an alternative that supports visualizing the magnitude of flows in activities network (HOEKSTRA; GROTH, 2014). 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 (ACUÑA, 2015; HUQ; APERS; WOMBACHER, 2013b; LERNER; BOOSE, 2014b; MCPHILLIPS et al., 2015b; MURTA et al., 2014) and others intend to be generic for any provenance application (HOEKSTRA; GROTH,

2014; KOHWALTER et al., 2016). Generic approaches use interoperable provenance formats (e.g., OPM, PROV), as discussed in Section 4.3.2.2. 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.

#### 4.3.3.3 Comparison

Some provenance approaches support comparing *data* to present differences between results (DAVISON, 2012) and for cache invalidation (GUO; ENGLER, D. R., 2010). Others support comparing *provenance* graphs to understand differences between executions (BAO et al., 2009; FREIRE et al., 2006). Since comparing general graphs is equivalent to the sub-graph isomorphism problem, which is NP-complete (SORLIN; SOLNON, 2005), some approaches reduce the complexity of the comparison by using the system context. The system context can indicate the lack of loops in graphs (FREIRE et al., 2006), the guarantee of well-formed loops for trials written in SPFL (series-parallel graph overlaid with well-nested forking and looping) (BAO et al., 2009), and other information that is specific to each provenance system.

#### 4.3.4 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 (GROTH; MILES; MOREAU, 2005; CHITTIMALLI; NAIK, 2014), binary program executions (DEMSKY, 2009; CHIRIGATI; SHASHA; FREIRE, 2013), operating systems (GUO; SELTZER, 2012; MUNISWAMY-REDDY et al., 2006), scientific workflow management systems (LIN et al., 2009; WOLSTENCROFT et al., 2013; FREIRE et al., 2006), and database systems (CHENEY; CHITICARIU; TAN, 2007). 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 (OUSTERHOUT, 1998). Provenance collection in these languages benefits from more informative static program analysis techniques than scripts (CHITTIMALLI; NAIK, 2014). 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 (DEMSKY, 2009; CHIRIGATI; SHASHA; FREIRE, 2013). 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 impose 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 (LIN et al., 2009; WOLSTENCROFT et al., 2013; FREIRE et al., 2006). 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 SWfMS 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. It allows scripts to treat activities as white-box operations and obtain more preci-

sion. 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 (WOLSTENCROFT et al., 2013), and some approaches propose combining SWfMS to external tools to fill the black-boxes (CHAPMAN; JAGADISH, 2010) (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 (PIMENTEL et al., 2018b). 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 (CHENEY; CHITICARIU; TAN, 2007). 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 4.3.1). 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.

## 4.4 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. It could lead to a disconnected component of a citation graph, which could concentrate only on a small niche. Note, however, that Jalali and Wohlin (2012) 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 indicates that our results did not concentrate in a small niche.

We considered only papers that we had access to their content and that matched our inclu-

sion 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 pre-dates the first related approach (BECKER; CHAMBERS, 1988), 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 to identify features and classify 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 difficulties to identify whether some approaches were scripting provenance approaches, binary provenance approaches, or just had the benefits of provenance collection without the intention of collecting provenance.

## 4.5 Update

The provenance of the taxonomy we proposed in this chapter can be described by the process we detailed in Section 4.2. However, after proposing the taxonomy, new approaches have been proposed. To systematically update the literature review, we followed the guidelines proposed by Felizardo et al. (2016), which consists of applying an exhaustive forward snowballing. By February 16th, 2021, we visited 638 papers and selected 33 papers matching the inclusion criteria.

Six papers extend or describe previously identified approaches. Lerner, Boose, and Perez (2018) update RDataTracker to collect provenance from R scripts transparently. They use the overriding strategy to add provenance collection statements in the code. Greff et al. (2017) update Sacred to add a web visualization tool for monitoring the execution of scripts and other features that are not related to provenance. Zhang et al. (2017) extend YesWorkflow to integrate it to DataOne RunManagers and combine its definition provenance with the R and MATLAB scripts' execution provenance. The other three papers (DA CRUZ; NASCIMENTO, 2019; MEYER, 2016; KERY, 2017) describe parts of their respective approaches in more detail. They do not include anything that changes the categorization we reported in the ACM Computing Surveys publication (PIMENTEL et al., 2019a).

Table 4.2 presents the selection of new approaches with their papers. In this table, we categorized the approaches following the same principle of Table 4.1: we attempted to under-

stand the purpose of provenance in these tools. For approaches that did not clearly specify the usage goals, we inferred by the proposed features. We found 27 papers referring to 17 new approaches, as we discuss in more detail below.

**Collection.** Most<sup>3</sup> approaches use *mandatory executable annotations* that target the *definition* internally for provenance collection. The *adapr* (Accountable Data Analysis Process in R) project (GELFOND et al., 2018), the approach proposed by Chapman et al. (2021), *DfAnalyzer* (SILVA et al., 2018a,b, 2020; DIAS, L. G. et al., 2020), *ModelKB* (GHARIBI et al., 2019), and *trackr* (BECKER; MOORE; LAWRENCE, 2019) use these annotations to *instrument* the script and collect *execution provenance*. *DfAnalyzer*, *ModelKB*, and *trackr* also use these annotations to collect definition provenance. *DfAnalyzer* lets users declare the definition provenance. *ModelKB* and *trackr* *parse* the script when the annotations are invoked to collect it *dynamically*. These annotations are also used by *ModelKB* and *trackr* to collect a *snapshot of deployment provenance*. Finally, *adapr* uses the annotations to collect the *deployment provenance continuously*.

*Flowgraph* (PATTERSON et al., 2017a,b, 2018; PATTERSON, 2020), *SMLD* (SecureMLDebugger) (HAN et al., 2020), and *Vamsa* (NAMAKI et al., 2020) collect the provenance from scripts transparently (i.e., they do not require changes on scripts). *SMLD* uses the *passive monitoring* strategy to collect *execution provenance*. Initially, *Flowgraph* (PATTERSON et al., 2017b) also used this strategy, but it was later updated to use the *overriding* strategy. *SMLD* and *Vamsa* collect *definition provenance* by *parsing* the source code *statically*. *SMLD* also collects a *snapshot* of modules as *deployment provenance*.

Other IDE-based (*TRACTUS* (SUBRAMANIAN; MAAS; BORCHERS, 2020)) and notebook-based approaches (*Dataflow Notebook* (KOOP; PATEL, 2017), *nbgather* (HEAD et al., 2019; HEAD, 2020), *JuNEAU* (IVES et al., 2019; ZHANG; IVES, 2019, 2020), *ProvBook* (SAMUEL; KÖNIG-RIES, 2018), *Verdant* (KERY; MYERS, 2018; KERY et al., 2019), and *Wrattler* (PETRICEK; GEDDES; SUTTON, 2018)) also collect provenance transparently. However, instead of using the default overriding or passive monitoring strategies that operate internally on the script, they use these strategies at a higher level to detect the execution of code snippets or cells. *JuNEAU*, *nbgather*, *TRACTUS*, *Verdant*, and *Wrattler* use the passive monitoring strategy. *JuNEAU* also uses the passive monitoring strategy to trace the cell execution internally. *Dataflow Notebook* and *ProvBook* use the overriding strategy. Some of these approaches also collect *definition provenance* by *parsing dynamically* (SUBRAMA-

---

<sup>3</sup>In this section, we are considering only the approaches in the update for conciseness. For a discussion of the approaches we obtained before, check the ACM Computing Surveys publication (PIMENTEL et al., 2019a).

Table 4.2: Selected approaches in the update with provenance support: main and secondary goals. Labels in secondary goals column refer to goals: *Cache* – Caching; *Compr* – Comprehension; *Frame*—Framework; *Manag* – Management; *Repro* – Reproducibility.

Approach	Main goal	Secondary goals				
		Cache	Compr	Frame	Manag	Repro
adapr (GELFOND et al., 2018)	Reproducibility	✗	✗	✗	✗	✓
Albireo (WENSKOVITCH et al., 2019)	Comprehension	✗	✓	✗	✗	✗
Chapman et al. (2021)	Comprehension	✗	✓	✗	✗	✗
Dataflow Notebook (KOOP; PATEL, 2017)	Reproducibility	✗	✗	✗	✗	✓
DFAnalyzer (SILVA et al., 2018a,b; DIAS, L. G. et al., 2020; SILVA et al., 2020)	Comprehension	✗	✓	✓	✗	✗
Flowgraph (PATTERSON et al., 2017a,b, 2018; PATTERSON, 2020)	Comprehension	✗	✓	✗	✗	✓
JuNEAU (IVES et al., 2019; ZHANG; IVES, 2019, 2020)	Management	✗	✓	✗	✓	✗
ModelKB (GHARIBI et al., 2019)	Management	✗	✗	✗	✓	✓
nbgather (HEAD et al., 2019; HEAD, 2020)	Reproducibility	✗	✗	✗	✗	✓
Osiris (WANG, Jiawei et al., 2020)	Reproducibility	✗	✗	✗	✗	✓
ProvBook (SAMUEL; KÖNIG-RIES, 2018)	Reproducibility	✗	✗	✗	✗	✓
SMLD (HAN et al., 2020)	Comprehension	✗	✓	✗	✓	✓
trackr (BECKER; MOORE; LAWRENCE, 2019)	Management	✗	✓	✗	✓	✓
TRACTUS (SUBRAMANIAN; MAAS; BORCHERS, 2020)	Comprehension	✗	✓	✗	✗	✗
Vamsa (NAMAKI et al., 2020)	Comprehension	✗	✓	✗	✗	✗
Verdant (KERY; MYERS, 2018; KERY et al., 2019)	Management	✗	✓	✗	✓	✓
Wrattler (PETRICEK; GEDDES; SUTTON, 2018)	Reproducibility	✗	✗	✗	✗	✓
<b>Main Goal / Total</b>		0 / 0	7 / 10	0 / 1	4 / 5	6 / 11

NIAN; MAAS; BORCHERS, 2020; KOOP; PATEL, 2017; KERY; MYERS, 2018; KERY et al., 2019; IVES et al., 2019; ZHANG; IVES, 2019, 2020; PETRICEK; GEDDES; SUTTON,

2018; HEAD et al., 2019; HEAD, 2020) or *reading dynamically* (SAMUEL; KÖNIG-RIES, 2018). In addition to the overriding strategy, Dataflow Notebook uses *parseable internal annotations* that describe the dependencies among cells and identify these dependencies after each cell execution.

Albireo (WENSKOVITCH et al., 2019) and Osiris (WANG, Jiawei et al., 2020) infer the *execution provenance* of notebooks using the *post-mortem* strategy. They collect the *definition provenance* from existing notebooks by *parsing statically* and attempt to infer relationships among cells. Albireo reports these relationships for comprehension. Osiris attempts to reorder the cells to a sensible order for reproducibility.

Flowgraph and Vamsa use *external parseable annotations* that target *provenance*. These annotations are stored as patterns in knowledge databases, and these approaches match the patterns to the collected provenance.

**Management.** DfAnalyzer, JuNEAU, and SMLD store the provenance in relational databases. In addition to relational databases, JuNEAU uses a graph database and an object key-value store. Chapman et al. (2021) use a NoSQL database. Wrattler stores the provenance in a content database. The adapr project, Flowgraph, ModelKB, and trackr store the provenance in proprietary files or content databases. The approaches that store provenance in files use these files for sharing. DfAnalyzer and Chapman et al. (2021) export the provenance to interoperable PROV. DfAnalyzer, SMLD, and ModelKB also provide Web servers that can be used to share the provenance remotely.

Albireo, Dataflow Notebook, TRACTUS, and Vamsa only store the provenance in the transient memory and do not export it. Despite not exporting the provenance, Dataflow Notebook provenance is stored within notebooks in the form of parseable annotations. ProvBook and nbgather also keep the provenance in the transient memory by default, but they support exporting it to other files. ProvBook exports the provenance to RDF files with PROV and other ontologies. Additionally, it provides operations for converting these files back to notebooks. The nbgather approach uses program slicing on the provenance and exports clean and ordered notebook files. Similarly, Osiris shares ordered versions of notebooks that it creates by inferring dependencies in the provenance.

For *versioning*, ModelKB and SMLD track the *sequence* of execution. Verdant and nbgather track the evolution of cells in notebooks. However, nbgather considers a single execution session, while Verdant collects the evolution across sessions. Verdant stores the provenance in Git repositories and indicate the *intention* of the evolution. Similarly, adapr supports Git reposi-

tories and considers the *intention*, but instead of using it by default, it provides functions for committing and merging the provenance.

**Analysis.** For visualization, approaches that share interoperable files (CHAPMAN et al., 2021; SILVA et al., 2018a,b, 2020; SAMUEL; KÖNIG-RIES, 2018; DIAS, L. G. et al., 2020) can open these files in external tools. Dataflow Notebooks, DfAnalyzer, JuNEAU, ModelKB, nbgather, trackr, and Verdant show textual logs with the provenance. DfAnalyzer also displays data graphs. Albireo, TRACTUS, SMLD, Flowgraph, and adapr display process graphs. Vamsa and Flowgraph display combined graphs. DfAnalyzer, Vamsa, and SMLD support filtering the graphs. Albireo and Flowgraph support clustering. nbgather supports comparing the provenance. ModelKB supports comparing the data of multiple executions.

For querying, approaches that use database systems support their respective generic querying languages (SILVA et al., 2018a,b, 2020; HAN et al., 2020; CHAPMAN et al., 2021; IVES et al., 2019; ZHANG; IVES, 2019, 2020). As specific queries, Albireo, DfAnalyzer, trackr, SMLD, and ModelKB provide web interfaces; JuNEAU, nbgather, ProvBook, and Verdant support queries in the notebook web interface; Chapman et al. (2021), and trackr define functions.

Finally, Wrattler does not have an interface for analyzing the provenance. Still, it analyses the provenance to detect dependencies among cells in a notebook and uses these dependencies to automatically re-execute cells when one of their dependencies has changed.

## 4.6 Discussion

In this chapter, we propose a taxonomy to characterize approaches that collect provenance from scripts. We constructed the taxonomy through 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.

Regarding the taxonomy branches, we identified approaches that employ all mechanisms of provenance collection. However, few approaches collect fine-grained execution provenance in a transparent way (i.e., without demanding changes on the script). The only transparent approaches that collect fine-grained provenance are Albireo (WENSKOVITCH et al., 2019), the approach proposed by Becker and Chambers (1988), nbgather (HEAD et al., 2019; HEAD, 2020), CXXR (RUNNALLS, A.; SILLES, C., 2012; SILLES; RUNNALLS, 2010), the one proposed by Michaelides et al. (2016), Osiris (WANG, Jiawei et al., 2020), RDataTracker

(LERNER; BOOSE; PEREZ, 2018), Vamsa (NAMAOKI et al., 2020), and Wrattler (PETRICEK; GEDDES; SUTTON, 2018).

Albireo, Osiris, and Vamsa infer dependencies among variables statically by analyzing the source code structure of scripts and notebooks files (i.e., after their execution). nbgather uses a similar approach of analyzing dependencies among variables in cells, but it uses the approach dynamically during the notebook execution. Still, it only considers the code definition but not the variable values.

Becker and Chambers (1988) collect commands and variables in S (CHAMBERS, 1998), CXXR collect commands and variables in R. These approaches indicate only the actual dependencies among variables, but they do not collect values.

RDataTracker collects commands, variables, and values in R. Michaelides et al. (2016) collect block variables, values, and block calls in Blockly. Wrattler collects variables, values, data frames, and operations in Python, R, and javascript. Despite collecting variables and values, these approaches consider data structures as coarse-grained objects and collect full snapshots of them, even when only a minimal part changes.

Wrattler also has some support for fine-grained collection of changes in data frames, but it does not extend the support to other data structures. Chapman et al. (2021) and JuNEAU (IVES et al., 2019; ZHANG; IVES, 2019, 2020) also support fine-grained collection of changes on data structures, but they do not consider other types of variables.

Hence, research is needed to develop efficient fine-grained provenance collection that supports generic complex data structures.

Some approaches use version control systems to store and track the evolution of provenance. While 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 (MATTOSSO et al., 2010). Hence, research is needed to understand the evolution of provenance considering the exploratory nature of experiments. The closest approaches that try to overcome these issues are Variolite (KERY; HORVATH; MYERS, 2017; KERY, 2017), and Verdant (KERY; MYERS, 2018; KERY et al., 2019). These approaches define custom version models for collecting variants in parts of scripts and notebooks.

Regarding provenance analysis, few approaches support provenance clustering (ACUÑA; CHOMILIER; LACROIX, 2015; EICHINSKI; ROE, 2016; HUQ; APERS; WOMBACHER, 2013a; LERNER; BOOSE, 2014b; TARIQ; ALI; GEHANI, 2012; WENSKOVITCH et al., 2019; PATTERSON et al., 2017a,b, 2018; PATTERSON, 2020), and fewer of them support

---

graph filtering (LERNER; BOOSE, 2014b; SILVA et al., 2018a,b; TARIQ; ALI; GEHANI, 2012; DIAS, L. G. et al., 2020; SILVA et al., 2020; HAN et al., 2020; NAMAKI et al., 2020). It indicates an opportunity to propose different summarization techniques. Moreover, the current provenance graphs are limited to directed graphs representing the provenance as-is. However, in the context of scripts, distinct graphs that consider the structure of scripts could be considered.

# Chapter 5

## Provenance in Scripts

### 5.1 Introduction

Murta et al. (2014) propose noWorkflow as an approach to capture definition, deployment, and execution provenance from Python scripts transparently (i.e., without requiring users to change the script). noWorkflow uses provenance for comprehension, reproducibility, and experiment management.

The initial version of noWorkflow (we call it noWorkflow 0) traverses the AST to collect functions as definition provenance (i.e., *parsing statically*). It collects environment variables and uses the *modulefinder* module to collect imported modules as a *snapshot* of deployment provenance. For execution provenance, noWorkflow 0 applies *overriding and passive monitoring* strategies. It overrides the *open* function to capture all file accesses, and it traces the execution with a Profiler to obtain executed functions with parameters and return values. Thus, it captures provenance at an intermediate-grain. It is not as coarse-grained as approaches that consider only file derivations (ACUÑA; LACROIX; BAZZI, 2015; MUNISWAMY-REDDY et al., 2006), nor as fine-grained as approaches that consider variables (LERNER; BOOSE, 2014b) or bytes (DEMSKY, 2009).

noWorkflow 0 stores file definitions in a *content database* structure using SHA1 hash codes and it uses their metadata related with execution provenance in a *relational SQLite* database. Thus, it supports *generic SQL queries* for analyses. In addition to SQL queries, noWorkflow 0 provides a series of *specific* command-line operations for listing and *comparing* trials, activations, modules, and environment variables. It also provides a command to *export* a trial provenance to Prolog with predefined rules, allowing users to run Prolog queries. noWorkflow 0 also supports loading provenance in an *internal* visualization tool that presents an activation

graph for visual analysis. The activation graph presents a *clustered* sequence of activations in a trial (i.e., it is a *process graph*).

In addition to collecting, storing, and analyzing provenance from a trial, noWorkflow 0 supports a weak form of provenance evolution. Like CPL (MACKO; SELTZER, 2012) and ESSW (FREW; BOSE, 2001), it creates a *sequential* identifier for each trial and supports listing trials and comparing basic trial information, such as parameters, duration, and file accesses.

This thesis extends and replaces most parts of noWorkflow 0 to introduce fine-grained provenance collection, provenance collection on interactive notebooks (Chapter 6), content database packing, full versioning support, provenance sharing, and additional ways to query, visualize, and filter provenance. These features support not only provenance comprehension, but can also support applications in other areas. For instance, provenance versioning and sharing can support reproducibility. Moreover, fine-grained provenance collection supports debugging.

The extensions of noWorkflow resulted in two new versions with distinct features and provenance collection modes: noWorkflow 1 and noWorkflow 2. Despite refactoring most code of noWorkflow 0, noWorkflow 1 is a direct evolution of it that uses the existing infrastructure to add new features, such as fine-grained provenance collection, graph comparison, history navigation, Jupyter integration, and others. On the other hand, noWorkflow 2 is a very different implementation that rethinks many aspects of the previous versions. noWorkflow 2 implements a very different provenance collection that is less intrusive and more precise by using AST transformations instead of profiling and tracing. It also re-implements graph summarizing and matching with more efficient algorithms. It has a seamless Jupyter integration that collects dependencies across cells. Finally, noWorkflow 2 has more space-efficient provenance storage.

This chapter is structured in four sections, besides this introduction, according to the taxonomy we proposed in Chapter 4. Section 5.2 presents our approaches for provenance collection. Section 5.3 presents our approaches for provenance management. Section 5.4 presents our approaches for analysis. Finally, Section 5.6 concludes this chapter with final remarks.

This chapter describes approaches from papers we published in the International Provenance and Annotation Workshop (PIMENTEL et al., 2016a,b,c, 2018b), and Proceedings of the VLDB Endowment (PIMENTEL et al., 2017).

## 5.2 Provenance Collection

In this section, we describe how all versions of noWorkflow collect provenance according to the taxonomy we proposed in Chapter 4. None of the versions use annotations for collecting provenance from Python scripts. They work with general Python scripts. For collecting provenance with noWorkflow, users just need to install it and invoke their scripts using the noWorkflow command `now run script.py` instead of `python script.py`.

### 5.2.1 Definition Provenance

All noWorkflow versions use the Python *ast* module to *parse* and collect scripts, function definitions, global variables, arguments, and function calls as definition provenance, *statically*. noWorkflow 1 extends the noWorkflow 0 collections by collecting class definitions and the compiled script bytecode as well. It does not store the bytecode, as it only uses it to support fine-grained execution provenance collection, as we explain in Section 5.2.3.

While the original implementation is good enough to describe the scripts' definitions and present which definitions they have, it suffers from three problems. First, it only collects the definition provenance of the main script. Due to the database organization, it does not collect the definition provenance of local modules that may also be part of the experiment definition. Instead, these modules appear as deployment provenance, without information about function definitions and arguments. Second, it does not collect the position of global variables, arguments, and function calls due to Python limitations on the *ast*<sup>1</sup>. Finally, it does not relate definition provenance to execution provenance. Thus, noWorkflow 0 and 1 can only guess that an activation relates to a function definition based on its name, and they are susceptible to false positives.

For these reasons, we ditched the original definition provenance collection, and we propose a new one for noWorkflow 2. The new system's main goal is to have a generic representation of the definition provenance with precise locations and connect the definition provenance both to the execution provenance and the deployment provenance.

We use two concepts to represent the definition provenance in noWorkflow 2: code component and code block. A code component is an element that appears in the source. It can represent the script itself, literals, variables, arguments, function calls, function definitions, class definitions, or any other expression or statement that appears in the *ast*. All code components

---

<sup>1</sup>Python 3.8 added the position to *ast* nodes. However, this change is not backward compatible, and noWorkflow 0 and 1 do not support this Python version.

have positions indicating the line and column of its first and last character in the script. A code component can be composed of other code components. For instance, in the operation  $x + 1$ , we have a code component representing the sum, and this code component is composed of two code components that represent  $x$  and  $1$ .

In addition to the composition, code components belong to code blocks. A code block is a code component that contains a block of code. We consider scripts, function definitions, and class definitions as code blocks. Since code blocks are code components, they have all the information that exists on code components, in addition to the code snippet and the documentation. However, scripts do not belong to other code blocks and their positions represent the first character and the last character in the script, respectively.

This model integrates well with both the deployment provenance and the execution provenance. For deployment provenance, instead of collecting module scripts as a separate entity, we can now collect scripts as code blocks and associate them with the deployment provenance. It allows us to collect the definition provenance of local modules dynamically during imports. We associate activations to code blocks and evaluations to code components for execution provenance, as we explain in Section 5.2.3. Hence, we can associate what is being executed with what appears in the source code.

As stated before, limitations on the Python *ast* module do not allow noWorkflow 0 and 1 to have precise information on the position of elements in scripts. Thus, we collect definition provenance in noWorkflow in two steps: first, we read the source code and enrich the *ast* with the precise position of the nodes. Then, we visit the *ast* and create the code components accordingly.

For enriching the *ast*, we propose a new library, PyPosAST<sup>2</sup>. This library uses the Python *tokenize* module to parse the source code collecting the position of all strings, attributes, numbers, operators, names, and parentheses. Then, it uses the *ast* module to parse and visit the *ast* using both the data that already exists on it (i.e., line information that appear in some nodes), the collected positions, and the Python syntax to enrich all nodes of the *ast* and to add special new nodes to represent textual elements, turning it into a mix of an Abstract and a Concrete syntax tree.

The new approach for definition provenance collection also has some limitations. First, it requires defining a visitor function for every *ast* element that we want to collect as a code component. Thus, new Python releases that change the *ast* require new PyPosAST and noWorkflow 2 releases. On the other hand, noWorkflow 1 suffered this issue in a higher intensity by using

---

<sup>2</sup><https://github.com/JoaoFelipe/pyposast>

the bytecode. Unlike the *ast*, the bytecode API is not stable, and changes occur not only on minor releases but also on patch releases.

Additionally, changes on the *ast* that just add new elements do not break the existing version. Instead, the new elements are just ignored by noWorkflow 2. Second, the enrichment of the *ast* requires the script to use UTF-8 as encoding. It is not a problem in most situations, but some scripts use other encodings. Finally, as it operates both with the script and the *ast*, it requires the script's source code. It cannot extract any information from modules that are imported as bytecode.

In addition to scripts and code elements, all versions of noWorkflow collect input files as definition provenance by *reading*. They override the *open* function for collecting file accesses *dynamically* (i.e., during the execution). In noWorkflow 0 and 1, the only overridden function is the built-in one. In noWorkflow 2, this has been extended to the open functions available at the modules *io*, *os*, and *codecs*.

## 5.2.2 Deployment Provenance

All noWorkflow versions collect a *snapshot* of environment variables at the beginning of the execution. In addition to the environment variables, they also collect information about the platform, such as the operating system, the Python version, the processor architecture, the host-name, and others. Similarly, noWorkflow 0 and 1 collect imported modules at the beginning of the execution. They use the Python built-in module *modulefinder* to go through all the declared modules in a script, importing them and discovering more modules. While it is fine to collect environment variables as a snapshot, doing it for modules has four major drawbacks.

First, *modulefinder* attempts to import every import that appears in a script, without considering conditional imports and imports that should only be executed upon the execution of a function. For instance, suppose a script that imports *Qt* if it is installed, or *Tkinter*, otherwise. By running *modulefinder* on this script, it will indicate that the script uses both *Qt* and *Tkinter*. Similarly, suppose a helper function for debugging that imports *pdb*, but that is not currently called, as the program does not have a bug. *modulefinder* will import it and indicate that the script is using *pdb*.

Second, once imported by *modulefinder*, Python keeps the module in the memory and does not attempt to import it again during the script's normal execution. This behavior not only increases memory usage, but can also cause bugs in experiments. For instance, an experiment that compares *Matplotlib* graphs before and after using *Seaborn* could have *Matplotlib* graph

generations before and after importing *Seaborn*, since just importing it already changes the graphs aesthetics (YSEARKA, 2016). However, running *modulefinder* on this script before the execution would import *Seaborn* and change the expected behavior.

Third, the *modulefinder* only considers imports that use the Python syntax for importing. In addition to the syntax, Python also offers functions for dynamically loading and unloading imports by passing their names or paths as a string.

Finally, due to the high overhead of discovering and collecting all imported files at the beginning, the scripts require some time before starting to output any results, which may annoy some users who want fast results. In a demonstration, a viewer reported that he thought the script froze when we executed it with `noWorkflow`.

Thus, for `noWorkflow 2`, we propose a *continuous* approach for collecting modules. In this approach, we implement import hooks that are only called when the script attempts to import a module. With these hooks, we only collect modules that are in fact imported, we respect the order of execution, we can collect dynamic modules, and we distribute the collection overhead across the execution, allowing users to start to see results right after the beginning of the execution.

Implementing an import hook in Python is not as straightforward as using the *modulefinder*. We had to re-implement the Python import machinery that consists of a module finder and a module loader to collect the provenance and import the module as it normally does. After defining the finder, we added it to the highest priority of `sys.meta_path`, which is a variable designed to allow users to define how to import non Python files in Python.

Thus, while the new approach solves the drawbacks of the previous approach, it also includes a new one: it is more susceptible to break on Python updates. In fact, the finder API completely changed from Python 2 to Python 3, and we maintain two implementations of the import hook to support both versions.

### 5.2.3 Execution Provenance

As stated in Section 5.2.1, `noWorkflow` *overrides* the *open* functions to collect input files as definition provenance. We use the same override to collect intermediate and output files as execution provenance. The only change is the access mode.

In addition to file accesses, all `noWorkflow` versions collect function calls, parameters, and return values as execution provenance. `noWorkflow 0` and `1` use the *passive monitoring* strat-

egy by defining a Python Profiler that automatically receives `CALL` and `RETURN` events from Python and processing them to create function activations.

An activation follows a tree structure in which the parent activation represents the caller of its children activations. During the collection of activations, `noWorkflow` collects their execution time and their sequence, allowing it to describe how the program runs. It also collects their arguments and return values.

Since these events are entirely based on the program execution and Python is a dynamic language, associating the triggered events with the definition provenance is not trivial and may result in false positives (i.e., it receives an event that appears to match a function definition in the script, but it matches another function with the same name). For instance, suppose the intentionally simple implementation of the happy numbers problem (PORGES, 1945) presented in Figure 5.1. In this code, there are two functions named `show` (lines 4 and 15) and a call on line 23. During the definition provenance collection, `noWorkflow 0` and `1` collect the function definitions `show`, `process`, and the second `show` and assign ids to them, such as `F1`, `F2`, `F3`, respectively. Then, during the execution of the function `show` on line 24, they only receive a `CALL` event indicating that a function named `show` is being called, but the event does not specify which one. Thus, `noWorkflow 0` and `1` do not associate the executed function with their definition.

Hence, for `noWorkflow 2`, we replaced the passive monitoring strategy with an *overriding* strategy that transforms the *ast* during the execution and indicates how to collect the provenance of each code component and code block. During the transformation, we indicate which code component is going to execute for each expression.

Figure 5.2 presents parts of the transformed happy numbers script with function definitions and function call to help explaining the provenance collection through *ast* transformations. Note, however, that `noWorkflow 2` does not export this transformed script, as it compiles the transformed *ast* directly to *bytecode*, and some textual representations may not be faithful representations of the transformed *ast*.

In this script, observe the special variables `NOW` and `ACT`<sup>3</sup>. They represent the `noWorkflow` collector and the current function activation, respectively. The collector is an object that defines all the execution provenance collection methods that the transformer invokes. We use the current activation to build the activation tree by indicating the parent activation of each activation. In this case, we create the script activation, `ACT`, in line 2 by invoking `start_script`. Then,

---

<sup>3</sup>In the actual implementation we use the names `__noworkflow__` and `__now_activation__` for `NOW` and `ACT` to avoid collisions.

```
4 def show(number):
5     pass
6
7 def process(number):
8     while number >= 10:
9         new_number, str_number = 0, str(number)
10        for char in str_number:
11            new_number += int(char) ** 2
12        number = new_number
13    return number
14
15 def show(number):
16     if number not in (1, 7):
17         return "unhappy_number"
18     else:
19         return "happy_number"
20
21 final = process(n)
22 if DRY_RUN:
23     final = 7
24 print(show(final))
```

Figure 5.1: Intentionally simple implementation of the happy numbers problem.

in lines 69 and 72, we indicate that both the `print` and the `show` activations occur inside the script activation. Note however that the `ACT` arguments that appear in lines 12, 16, and 42 are not the same activation as the script activation. Instead, they are created by each function call and passed as parameters by the `function_def` decorator. This decorator is defined by a double call with two sets of parameters. The first set contains the parent activation and the code block id of the function definition. The second set contains a list of parameters, with their code component ids and default values. Both the code block id and the code component id are used to connect the execution provenance to the definition provenance.

Both the collection approach described for `noWorkflow 0` and `1` and the collection approach depicted in Figure 5.2 for `noWorkflow 2` represent coarse-grained execution provenance collections with activations and arguments. Coarse-grained provenance adopts the function activation order to infer the dependency among data, potentially leading to false-positive links. For instance, Figure 5.1 calls `show` after calling `process`, leading to the inference that the `show` result depends on the `process` result. In fact, this inference happens to be true when `DRY_RUN` is `False`. However, the same inference would lead to a false-positive result should the global variable `DRY_RUN` be `True`. It occurs because `final` would be assigned to `7`, which does not depend on the result of `process`.

For addressing this issue, both `noWorkflow 1` and `noWorkflow 2` support fine-grained provenance collection through different implementations. `noWorkflow 1` extends the provenance collection of `noWorkflow 0` by defining a Python Tracer (PIMENTEL et al., 2016c). A Tracer is usually intended to support the implementation of debuggers. It works like a Profiler, but instead of receiving only `CALL` and `RETURN` events, it also receives `LINE` events for each executed

```
2 ACT = NOW.start_script(__name__, S1)
3 try:
11     @NOW.function_def(ACT, F1) (('number', P1, None))
12     def show(ACT, number):
15     @NOW.function_def(ACT, F2) (('number', P2, None))
16     def process(ACT, number):
41     @NOW.function_def(ACT, F3) (('number', P3, None))
42     def show(ACT, number):
69     NOW.call(ACT, C4, print) (
70         NOW.argument(ACT, A4) (
71             NOW.call(ACT, C5, show) (
72                 NOW.argument(ACT, A5) (final)
73             )
74         )
75     )
76 except:
77     NOW.collect_exception(ACT)
78     raise
79 finally:
80     NOW.close_script(ACT)
```

Figure 5.2: Transformed script with function definitions and function call.

line. noWorkflow 1 uses these line events together with bytecode analysis to extract variables, variable usages, and dependencies among variables through program slicing (PIMENTEL et al., 2016c).

However, using a Tracer in combination with bytecode to collect provenance has many drawbacks. First, it does not handle operations split into multiple lines due to the nature of line events. Second, it is imprecise for lines with multiple operations since the analysis occurs line by line. Consequently, it is hard to collect provenance from collections and structured objects since accessing collection items and object attributes are independent operations. Third, the execution of external modules also triggers LINE events, decreasing the provenance collection performance when the users are only interested in their own execution. Finally, as stated before, the bytecode API is not stable and causes incompatibilities on Python updates.

Hence, for noWorkflow 2 we use a different approach for collecting fine-grained provenance. As an extension to the overriding strategy applied for coarse-grained collection (as presented before in Figure 5.2), we modify the AST to add operations to collect evaluations and dependencies among evaluations. An evaluation is the materialization of a value in a code component. For instance, a code component that represents an iterator name in a loop may have multiple evaluations representing each value assumed by the iterator.

A dependency between two evaluations has two main types: reference or derivation. A reference dependency indicates that an evaluation represents the same object as the object it was derived from. It is the type of dependency that appear on assignments. For instance, the

evaluation of `final` in line 23 of Figure 5.1 is a reference dependency of the evaluation of 7, meaning that they represent the same object. A posterior change on the object internal structure would affect both evaluations<sup>4</sup>.

On the other hand, a derivation dependency indicates that an evaluation was derived from another. It occurs on common unary and binary operations. For instance, in line 11 of Figure 5.1, the evaluation of `int(char) ** 2` has a derivation dependency to the evaluation `int(char)` and another derivation dependency to 2.

We use reference dependencies together with checkpoints to collect the provenance of collections and structured objects according to the Versioned-PROV model (PIMENTEL et al., 2018b). In this model, all evaluations connected by reference dependencies share the same members. A member is another evaluation that can represent an attribute of an object or an item of a collection. During the analysis phase, we use the checkpoints to reconstruct the state of each structured object at any given moment and infer the provenance. It allows a fast collection of the provenance from structured objects without requiring a recursive collection of all of their members and sub-members. It is also a way of supporting the provenance of recursive data structures. Versioned-PROV is one of the contributions of this work. We opted not to detail it in the body of the thesis since it is a very specialized technique for dealing with mutable data structures. However, it is available and evaluated in Appendix A.

Different from `noWorkflow 1` that collects the fine-grained provenance of variables, `noWorkflow 2` represents only evaluations in its data model. Hence, variable definitions and usages appear as distinct evaluations. For instance, in Figure 5.1, `final` in lines 21, 23, and 24 enacts three different evaluations when they are executed. For indicating that they are the same variable, during the execution, we keep a scope map on the Activation object that indicates for each function call, which evaluation represents the variable. Hence, in this example, during the execution of line 21, `noWorkflow 2` creates an evaluation for `final` and adds it to the scope of the script activation indicating that the name `final` was defined by this new activation. Then, during the execution of line 23, it creates a new evaluation for `final` and replaces the evaluation on the scope map. Finally, during the execution of line 24, it recognizes it as usage. Instead of replacing the variable in the scope map, it creates a reference dependency from the evaluation of `final` in line 24 to the evaluation of `final` in line 23, indicating that they represent the same variable and object. Thus, it is possible to indicate that the result of `show` (that receives `final`) does not depend on the result of `process`.

---

<sup>4</sup>It is not trivial to change the internal structure of numbers in Python, but it is possible to do using C extensions.

### 5.2.4 Summary

Table 5.1 compares provenance collection in noWorkflow versions (in the bottom) with the approaches we obtained in Chapter 4. Compared to noWorkflow 0, noWorkflow 1 adds the collection of fine-grained variables as execution provenance and extends the collection of definition provenance to elements that were not covered by the previous version.

noWorkflow 2 replaces the provenance collection mechanisms by exclusively adopting the overriding strategy for execution provenance collection and using import hooks for collecting modules as deployment provenance continuously. The move towards the overriding strategy also occurred in other approaches for achieving more control and transparency. RDataTracker used to require the instrumentation of scripts, and Flowgraph used to apply the passive monitoring strategy.

noWorkflow 2 is the only approach that collects provenance from generic complex data structures efficiently. Michaelides et al. (2016) and RDataTracker can collect provenance from structured variables, but they collect a snapshot of the whole structure at every collection. Chapman et al. (2021), Wrattler, and JuNEAU support efficient manipulation of data frames, but they do not support other structured data. noWorkflow 2 identifies changes on individual members of data structures, keeps references among objects that represent the same data, and uses the checkpoint of each member derivation to reconstruct complete states (PIMENTEL et al., 2018b).

Table 5.1: Provenance collection strategies. Labels in Annotations columns refer to categories described in Chapter 4 *Exte* —External; *Inte* —Internal; *Pars* —Parseable; *Exec* —Executable; *Incl* —Inclusive; *Excl* —Exclusive; *Defi* —Definition; *Prov* —Provenance; *Man* —Mandatory; *Opt* —Optional.

Approach	Granularity	Annotations					Execution				Depl.		Definition			
		Placement	Extraction	Inclusiveness	Target	Necessity	Passive Monitoring	Overriding	Post-Mortem	Instrumentation	Snapshot	Continuous	Reading	Parsing	Static	Dynamic
adapr	Modules, Files (I/O)	<i>Inte</i>	<i>Exec</i>	<i>Incl</i>	<i>Defi</i>	<i>Man</i>	X	X	X	✓	X	✓	X	X	X	X
Albireo	Cells, Variables	X	X	X	X	X	X	X	✓	X	X	X	✓	✓	X	
Astro-WISE	User defined, Attributes, Files (I/O), Parameters, Source	<i>Inte</i>	<i>Exec</i>	<i>Incl</i>	<i>Defi</i>	<i>Man</i>	X	X	X	✓	X	X	✓	X	X	✓

Continued on next page

Approach	Granularity	Annotations					Execution				Depl.		Definition			
		Placement	Extraction	Inclusiveness	Target	Necessity	Passive Monitoring	Overriding	Post-Mortem	Instrumentation	Snapshot	Continuous	Reading	Parsing	Static	Dynamic
Becker and Chambers (1988)	Commands, Variables, Random Seed	X	X	X	X	X	X	✓	X	X	X	X	X	X	X	✓
Bochner, Gude, and Schreiber (2008)	User defined, Files, Platform	Inte	Exec	Incl	Defi	Man	X	X	X	✓	X	✓	✓	X	X	✓
Chapman et al. (2021)	Data frames	Inte	Exec	Incl	Defi	Man	X	X	X	✓	X	X	X	X	X	X
CPL	N/A	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CXXR	Commands, Variables, Random Seed, Files (I)	X	X	X	X	X	X	✓	X	X	X	X	X	X	X	X
Dataflow Notebook	Cells, Dependencies	Inte	Pars	Incl	Defi	Man	X	✓	X	✓	X	X	X	✓	X	✓
Datatrack	User defined, Parameters, Platform, Modules	Inte	Exec	Incl	Defi	Man	X	X	X	✓	X	✓	X	X	X	X
DFAnalyzer	User defined	Inte	Exec	Incl	Defi	Man	X	X	X	✓	X	X	✓	X	X	✓
ES3	Files (I/O - metadata)	X	X	X	X	X	X	✓	X	X	X	X	X	X	X	X
ESSW	User defined, Processes, Files (I/O)	Inte	Exec	Incl	Defi	Man	X	X	X	✓	X	X	✓	X	X	✓
Flowgraph	Functions, Files (I/O), Stack Trace	Exte	Pars	Incl	Prov	Opt	✓	✓	X	X	X	X	X	X	X	X
IncPy	Functions, Globals, Stack, Output, Files (I/O)	Inte	Exec	Incl Excl	Defi	Opt	X	✓	X	✓	X	X	✓	✓	✓	✓
JuNEAU	Cells, Data frames	X	X	X	X	X	✓	X	X	X	X	X	X	✓	X	✓
Lancet	Arguments, Commands, Platform, Env. Var.	Inte	Exec	Incl	Defi	Man	X	X	X	✓	✓	X	X	✓	✓	X
Magni	User defined, Stack Trace, Platform, Source	Inte	Exec	Incl	Defi	Man	X	X	X	✓	X	✓	✓	X	✓	X

Continued on next page

Approach	Granularity	Annotations					Execution				Depl.		Definition			
		Placement	Extraction	Inclusiveness	Target	Necessity	Passive Monitoring	OVERRIDING	Post-Mortem	Instrumentation	Snapshot	Continuous	Reading	Parsing	Static	Dynamic
Michaelides et al. (2016)	Blocks, Calls, Random Seed, Values, User Input	X	X	X	X	X	X	✓	X	X	X	X	X	X	X	✓
ModelKB	Files (I/O), Arguments, Models	Inte	Exec	Incl	Defi	Man	X	X	X	✓	✓	X	X	✓	X	✓
nbgather	Cells, Variables, Dependencies	X	X	X	X	X	✓	X	X	X	X	X	✓	X	✓	✓
Osiris	Cells, Variables	X	X	X	X	X	X	X	✓	X	X	X	✓	✓	✓	X
ProvBook	Cells	X	X	X	X	X	X	✓	X	X	X	X	✓	X	X	✓
Provenance Curious	Language Constructs, Files (I/O)	Exte	Pars	Incl	Defi	Man	X	X	✓	X	X	X	✓	✓	✓	X
pypet	Arguments, Output, Sumatra	Inte	Exec	Incl	Defi	Man	X	X	X	✓	✓	X	✓	X	✓	X
					Prov	Opt										
RDataTracker	Commands, Variables, Values, Env. Var., Platform, Modules, Files (I/O)	X	X	X	X	X	✓	✓	X	X	✓	✓	✓	X	X	✓
RFlow	User defined, Files (I/O), Source	Exte	Pars	Incl	Prov	Man	X	X	X	✓	X	X	✓	X	✓	X
Sacred	User defined, Output, Modules, Host, Source, Files (I/O)	Inte	Exec	Incl	Defi	Opt	X	✓	X	✓	✓	X	✓	X	✓	✓
SMLD	Functions, Platform, Modules	X	X	X	X	X	✓	X	X	X	✓	X	X	✓	✓	X
SPADE	Functions, Returns, Arguments, Stack Trace, Env. Var.	Exte	Pars	Excl	Defi	Opt	X	✓	X	X	✓	X	X	X	X	X
StarFlow	Functions, Modules, Files (I/O), Stack Trace	Inte	Pars	Incl	Defi	Opt	✓	✓	X	✓	✓	X	✓	✓	✓	✓
			Exec													
Sumatra	Modules, Files (I/O)	Exte	Pars	Incl	Prov	Opt	X	X	✓	X	✓	X	✓	X	✓	X
trackr	Values, Modules	Inte	Exec	Incl	Defi	Man	X	X	X	✓	✓	X	✓	X	✓	✓
TRACTUS	Function calls	X	X	X	X	X	✓	X	X	X	X	X	✓	X	✓	✓
Vamsa	Language Constructs, Variables	Exte	Pars	Incl	Prov	Opt	X	X	X	X	X	X	✓	✓	✓	X

Continued on next page

Approach	Granularity	Annotations					Execution				Depl.		Definition			
		Placement	Extraction	Inclusiveness	Target	Necessity	Passive Monitoring	OVERRIDING	Post-Mortem	Instrumentation	Snapshot	Continuous	Reading	Parsing	Static	Dynamic
Variolite	Arguments, Output, Source	Exte	Pars	Incl	Prov	Opt	✗	✗	✓	✗	✗	✗	✓	✗	✓	✗
VCR	User defined, Values, Calls, Stack Trace	Inte	Exec	Incl	Defi	Man	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
Verdant	Cells	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓
versuchung	User defined, Files (I/O), Source	Inte	Exec	Incl	Defi	Man	✗	✗	✗	✓	✗	✗	✓	✗	✓	✓
WISE	Processes, Modules, Files (I/O - metadata)	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓	✗	✓	✗	✓	✗
Wrattler	Variables, Dependencies, Data frames	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓
YesWorkflow	User defined	Inte Exte	Pars	Incl	Defi	Man	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗
noWorkflow 0	Functions, Env. Var., Platform, Modules, Files (I/O)	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓	✗	✓	✓	✓	✓
noWorkflow 1	Functions, Variables, Env. Var., Platform, Modules, Files (I/O)	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓	✗	✓	✓	✓	✓
noWorkflow 2	Functions, Variables, Complex Data Structures, Env. Var., Platform, Modules, Files (I/O)	✗	✗	✗	✗	✗	✗	✓	✗	✗	✓	✓	✓	✓	✓	✓

## 5.3 Provenance Management

In this section, we describe how noWorkflow manages provenance.

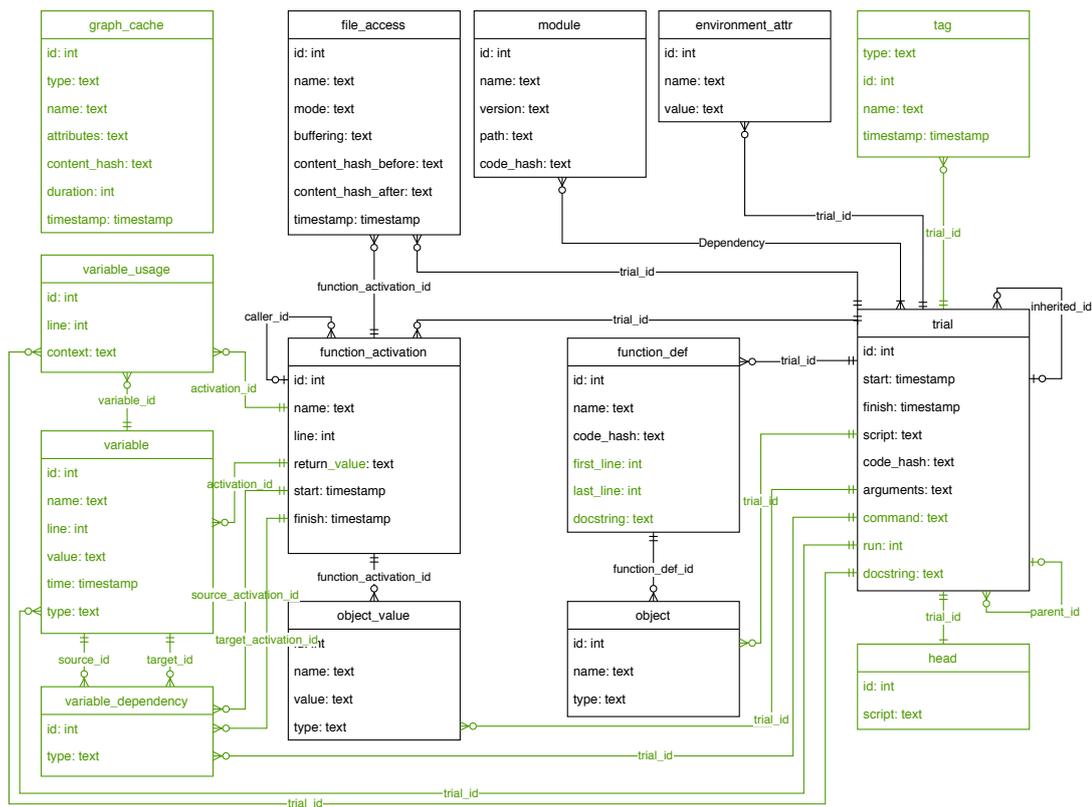


Figure 5.3: noWorkflow 1 relational data model. Green represent additions.

### 5.3.1 Storage

All versions of noWorkflow store provenance in a relational SQLite database and a content database. When the user runs noWorkflow for the first time in a directory, it creates a `.noworkflow` sub-directory with both databases. In the content database, noWorkflow stores scripts, function definitions, and read and written files. In the relational database, it stores structured provenance data and organizes the relationships and metadata of the data stored in the content database. It uses code hashes to refer to files in the content database.

**Relational Database.** Figure 5.3 presents the noWorkflow 1 data model in comparison to noWorkflow 0. As stated before, noWorkflow 1 is a direct evolution of noWorkflow 0. Its data model is just an extension with the addition of new tables and columns. Elements in green represent additions.

In this model, a `trial` represents an execution of an experiment. It stores the code hash of the main script as definition provenance. Besides the main script in the `trial` table, it also stores definition provenance in the tables `function_def` (function definitions) and `object` (global variables, arguments, and function call definitions).

For deployment provenance, it stores imported libraries in the `module` table and environment variables in the `environment_attr` table.

As execution provenance, noWorkflow 0 and 1 use the tables `function_activation` (function activations and main script execution), `object_value` (actual values of global variables and arguments), and `file_access` (accessed files). For fine-grained provenance, noWorkflow 1 also stores execution provenance in the tables `variable` (variable bindings), `variable_usage` (usage of variables), and `variable_dependency` (dependencies among variables).

In addition to these tables, noWorkflow 1 has other changes in the data model to support versioning and optimizations. For versioning, it includes the table `head` for navigating the history, the table `tag` to assign names to versions, and the attribute `parent_id` in `trial` to keep the history. For optimizations, it transformed global auto-increment `id` columns into primary keys composed of the pair `trial_id`, `id`, with the `id` being managed by the execution – in some tables, it was necessary to add `trial_id` columns. As a consequence, insertions are faster, and it is possible to query elements from a trial without many joins. Additionally, noWorkflow 1 uses the table `graph_cache` to store processed visualizations and speed-up the exhibition of them.

The data model of noWorkflow 1 has some issues. First, it does not relate execution provenance to definition provenance. Even though functions stored in the `function_def` table are activated and stored in the `function_activation` table, there is no relationship between these tables beside the function name, which is not a unique identifier. The same issue occurs between `object_value` and `object`. Second, multiple tables (`module`, `function_def`, and `trial`) have code hashes referring to similar code elements in the content database. Ideally, these references could be in a single table. Third, `object_value` and `variable` store similar elements, and some elements are stored twice for compatibility between fine-grained collection and coarse-grained collection. Finally, some names could be more generic (e.g., `function_activation` and `function_def`) to support other types of elements.

For solving these issues and others, noWorkflow 2 uses a different data model, as presented in Figure 5.4. In this model, there is a direct relationship between the execution provenance (activation, evaluation) and the definition provenance (`code_block`, `code_component`). In the new model, an activation is an evaluation of a function call in most cases. Still it can also be an evaluation of a class definition or the main script itself. An evaluation that is not an activation can be any element in the code that produces a value (`repr`) at a given moment (`checkpoint`). Hence, it combines and replaces the `variable` and

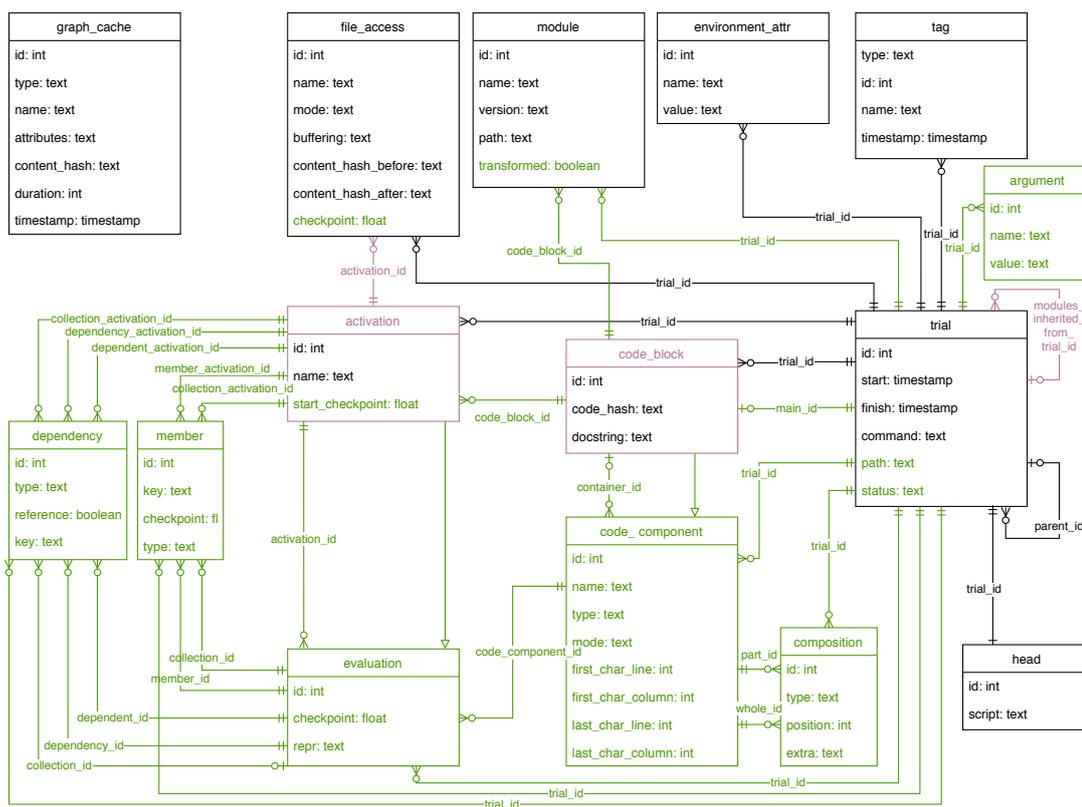


Figure 5.4: noWorkflow 2 relational data model. Green represent additions and semantic changes. Purple represents renames. It does not show removals.

object\_value tables. The tables `dependency` and `member` represent dependencies and memberships among evaluations as described in Section 5.2.3.

A `code_block` is a `code_component` that has a block of code, such as a script definition, a class definition or a function definition. We store these elements in the content database. Other types of `code_components` are any textual element that appears at any position in the code. Hence it stores the position. We use the table `composition` to rebuild the AST from `code_component`.

Since `code_block` stores code elements and has a code hash referring to the content database, we transformed the imported modules and the main script definition into code blocks, removing the `code_hash` and `docstring` columns from `trial` and `module`. Additionally, the `module` table previously had a N\*N relationship with `trial`. For this change, we replaced it with a 1\*N relationship that made the execution faster at the expense of using more space for repeated modules.

The data model of noWorkflow 2 allows answering some provenance queries that the data model of noWorkflow 1 does not. Since noWorkflow 2 relates definition provenance to execution provenance, it is possible to answer where a given value was obtained in the code. While

in noWorkflow 1 it can be loosely inferred by the name of variables – which may be defined in multiple locations ambiguously – noWorkflow 2 allows precise identification of the location of the evaluation that produced a value. Unlike noWorkflow 1, noWorkflow 2 can also identify function calls and other operations (e.g., `sum`, collection access, object attribute) as the origin of a given value.

Similarly, noWorkflow 1 identifies activations by their definition names (i.e., names stored in their `__name__` attributes by the `def` statement). However, Python uses functions as first-class citizens and allows the redefinition of function names (e.g., `number = int; number('3')` are valid statements). The concepts of evaluation and activation in noWorkflow 2 allows the identification of such situations and querying for which functions have different invocation names (i.e., it is possible to know that the evaluation `number('3')` represents the activation of `int`).

Finally, the definition of `members` in the noWorkflow 2 data model following the Versioned-PROV model allows querying components of complex data structures. These components can answer queries such as "What was the value of the element in the  $n^{th}$  position of a list?", "Which evaluations lead to the generation of sub-components?", and "What was the type of the evaluation?". Note for the last question that everything is an object in Python with an attribute `__type__` indicating their classes. We use the Versioned-PROV model for collecting these attributes.

**Content Database.** noWorkflow 1 and 2 supports two types of content databases, which users can select per project. The original one (i.e., from noWorkflow 0) stores copies of the files without compression. When it collects a file, noWorkflow computes the SHA1 hash of it and stores it using the same strategy Git uses for storing files: it uses the first two digits of the hash as the directory name in the content database and the remaining digits as the filename inside this directory. For reading a file, it uses the SHA1 hash and accesses the file directly.

This strategy is easy to implement and provides fast accesses, but it does not give proper treatment for big files or files that grow along with several trials. Small changes in a file results in changes in the SHA1 hashes and separate storage of file revisions.

The other type of content database uses Git itself for storing files (PONTES, 2018). Git reduces the storage overhead using two techniques. It not only compresses files with zlib (GAILLY; ADLER, 2017), but also combines objects into packfiles that contain one version of them and deltas from one version to another (CHACON; STRAUB, 2014). Thus, it can reconstruct big files that received small modifications over time.

Using Git commands for interacting with Git would impose a significant performance overhead due to the excess of system calls. Instead, noWorkflow uses the libraries *Dulwich* (VERNOOJJ, 2018) or *PyGit2* (IBÁÑEZ et al., 2018) to perform Git operations. While the former is a pure Python implementation of Git (i.e., slower but easier to install), the latter is a library that provides bindings for a C library (i.e., faster but harder to install). Hence, when both libraries are installed, noWorkflow uses *PyGit2*.

In our tests (PONTES, 2018) (see Annex A), the Git content database reduces the size overhead by 65.23% on average with an extra processing time overhead of 1.90%, compared to the noWorkflow 0 content database. Additionally, the Git content database allows the execution of a garbage collection to force the creation of `packfiles` that further reduce the storage overhead, reaching a reduction of 73.79% when compared to the noWorkflow 0 content database.

### 5.3.2 Sharing

As stated before, noWorkflow stores both the content database and the relational database in a `.noworkflow` sub-directory. Thus, the easiest way to share the collected provenance with all trials is to share this directory.

Besides the directory, all noWorkflow versions have a command to export *Logic Prolog* facts and rules (`now export`). This command exports rows from the relational database tables as compound terms composed of a *functor* based on the table name and *arguments* representing the columns. Figure 5.5 presents a subset of exported facts from a trial that represents the execution of the script depicted in Figure 5.1. In addition to Prolog facts, this command exports Prolog rules to help querying the data employing transitive closures.

For noWorkflow 1 and 2, we extended the sharing features. In both versions, we included a command to export a *Graph* file for the fine-grained dataflow in the *GraphViz* format (`now dataflow`), as we show in Section 5.4.2. In noWorkflow 2, we also added a command to export interoperable<sup>5</sup> Versioned-PROV (PIMENTEL et al., 2018b) files (`now prov`).

In addition to these *local* sharing methods, noWorkflow 1 and 2 also include a small *web* server (`now vis`). This web server aims to visualize the trial history, with their activation graphs, files, environment variables, and parameters, as we present in Section 5.4.2. However, users can also configure it to allow a *remote* sharing of the provenance.

---

<sup>5</sup>Versioned-PROV is not as interoperable as plain PROV at this moment. We intend to develop an algorithm for converting it to PROV in the future.

```

11 % FACT DEFINITION: trial(Id, Script, Start, Finish, Command,
12 %                               Status, ModulesInheritedFromTrialId, ParentId, MainId).
13 trial(1, 'script.py', 1610471126.1342, 1610471126.380694, 'run_script.py',
14     'finished', nil, nil, 1).

282 % FACT DEFINITION: code_component(TrialId, Id, Name, Type, Mode,
283 %                               FirstCharLine, FirstCharColumn,
284 %                               LastCharLine, LastCharColumn, ContainerId).
285 code_component(1, 1, 'script.py', 'script', 'w', 1, 0, 25, 0, nil).
286 code_component(1, 10, 'show', 'function_def', 'w', 4, 0, 5, 8, 1).
287 code_component(1, 17, 'number', 'param', 'w', 4, 9, 4, 15, 10).
288 code_component(1, 19, 'process', 'function_def', 'w', 7, 0, 13, 17, 1).
289 code_component(1, 26, 'number', 'param', 'w', 7, 12, 7, 18, 19).
290 code_component(1, 43, 'str(number)', 'call', 'r', 9, 36, 9, 47, 19).
291 code_component(1, 60, 'int(char)', 'call', 'r', 11, 26, 11, 35, 19).
292 code_component(1, 73, 'show', 'function_def', 'w', 15, 0, 19, 29, 1).
293 code_component(1, 101, 'process(n)', 'call', 'r', 21, 8, 21, 18, 1).
294 code_component(1, 115, 'print(show(final))', 'call', 'r', 24, 0, 24, 18, 1).
295 code_component(1, 119, 'show(final)', 'call', 'r', 24, 6, 24, 17, 1).

433 % FACT DEFINITION: code_block(TrialId, Id, CodeHash, Docstring).
434 code_block(1, 1, 'de77867d4de96c80e17161a9b4da7cafdc091fb0', '').
435 code_block(1, 10, '5fa2287e2051714c9897df7cae2674e5c715bafa', '').
436 code_block(1, 19, '24b8c944a0cb0a418a7ec6c4fe669a47b7c96721', '').
437 code_block(1, 73, '61663f8aa387cc1d3fbb9958bc44a786361c3ae9', '').

447 % FACT DEFINITION: activation(TrialId, Id, Name, StartCheckpoint, CodeBlockId).
448 activation(1, 1, '__main__', 0.17218730000058713, 1).
449 activation(1, 13, 'process', 0.17250679999960994, 19).
450 activation(1, 20, 'str', 0.17266079999899375, nil).
451 activation(1, 29, 'int', 0.172852800000328, nil).
452 activation(1, 36, 'int', 0.17296220000025642, nil).
453 activation(1, 52, 'print', 0.173218399999314, nil).
454 activation(1, 53, 'show', 0.17322749999948428, 73).

464 % FACT DEFINITION: evaluation(TrialId, Id, Checkpoint, CodeComponentId, ActivationId,
465 %                               Repr, MemberContainerId).
466 evaluation(1, 1, 0.1734820999990916, 1, 0, '<module__main__from_script.py>', 1).
467 evaluation(1, 13, 0.17312279999896418, 101, 1, '1', 29).
468 evaluation(1, 20, 0.17268610000064655, 43, 13, '10', 20).
469 evaluation(1, 29, 0.1728758999997808, 60, 13, '1', 29).
470 evaluation(1, 36, 0.17298379999920144, 60, 13, '0', 36).
471 evaluation(1, 52, 0.17345229999955336, 115, 1, 'None', 52).
472 evaluation(1, 53, 0.17337319999933243, 119, 1, 'happy_number', 61).

```

Figure 5.5: Subset of Prolog facts from a trial. We reordered lines and added line breaks when needed to fit the page.

### 5.3.3 Reproducibility

As stated in Chapter 4, provenance is useful for the reproducibility of experiments, as it allows scientists to share not only the findings but also the data, programs, and environments. For supporting these operations, noWorkflow provides the sharing features we described in Section 5.3.2.

Additionally, scientists can also use provenance for reproducibility by comprehending third-party experiments, and comparing different executions to check if a new trial could replicate the results of the previous one. noWorkflow supports both visualizing and comparing trials (see Section 5.4).

Finally, another aspect of reproducibility is being able to re-run previous trials under similar conditions. We added the command `now restore` to noWorkflow 1 and 2 to restore files of a previous trial for re-executions (PIMENTEL et al., 2016b). Users can run this command either passing a trial as the argument or a trial and a file path. When they pass both the trial and the file path, noWorkflow restores the specified file. However, when they pass only the trial, the command restores the Python script with its local libraries and all input files. It does not restore intermediate files, output files, nor external libraries. While not restoring external libraries may be a problem for an actual reproduction of the trial, we opted not so to avoid breaking the Python installation. Nonetheless, by running the reproduction with noWorkflow, it is still possible to compare the libraries for fixing the external dependencies using proper installation methods. We present an example of this operation in Section 5.3.4.

### 5.3.4 Versioning

All versions of noWorkflow generate *sequential trial ids*. These ids allow identifying which trial occurs before or after the other and support some basic trial comparisons. With the addition of the `now restore` command in noWorkflow 1 and 2 (see Section 5.3.3), we also extended the version model to a more formal definition that encompasses the *intention* of the trial evolution (PIMENTEL et al., 2016b). Additionally, we added the possibility of creating tags to reference trials instead of their sequential ids and created automatic semantic tags to describe the trial's intention. These automatic tags follow a semantic versioning schema based on three numbers: X.Y.Z. The semantic versions start as 1.1.1 for Trial 1. If the script is re-executed with the same code and input, it increments Z (i.e., automatic tag 1.1.2). If it uses the same code but a different input, it increments Y and resets Z to 1 (i.e., automatic tag 1.2.1). Finally, if the code changes, it increments X and resets both Y and Z to 1 (i.e., automatic tag 2.1.1). This versioning schema further describes the intention of the evolution.

**Version Model.** Conradi and Westfechtel (1998) state that a version model should define the organization of the version space (i.e., how a product is versioned) and the interrelation of the product space (i.e., how a product is structured) and the version space. We define our product space as an experiment, containing its scripts, data, execution traces, etc. The entry point of our product space is the main script of the experiment. We recursively capture imported modules from this script, accessed files during execution, and the execution provenance. Thus, we have scripts (including imports), input files, intermediate files, and output files as file objects. We identify file objects solely by their path within the experiment directory.

File objects describe the structure of the experiment: that is, all files needed by the experiment, which includes the script itself (definition provenance), imported modules (deployment provenance), and accessed (read/write) files (execution provenance). On the other hand, we also have logical provenance information that is not stored in files: functions called during execution, parameters values, variable values, etc. In our product space, we have a special object called *logical object* that contains all the aforementioned logical provenance information. This way, we can say that our product space comprises multiple file objects and one logical object.

Our *version space* (CONRADI; WESTFECHTEL, 1998) has two versioning levels: *trial version* (i.e., the trial id) and file object version. Trial versions represent the state of the experiment in terms of *file object* versions read or written within each trial, together with the logical object version produced by the trial. On the other hand, file object versions represent the state of file objects at each file access during the whole experiment execution (throughout all trials). File object versions may contain extra attributes (metadata) besides the state of file objects: modules may have their semantic versions declared by developers (e.g., 3.5.1), files may have their moment of opening and opening mode (read/write), etc.

We apply this distinction between trial versions and file object versions because scripts can write to some file objects more than once, generating more than one version of the file object within a single trial. Due to this distinction, our version space supports restoring trial versions as a whole, with all input file objects, or specific file object versions (e.g., an intermediate version of a file object). However, to restore a specific file object version, users should inform which object they want to restore individually and in which moment (i.e., by indicating a timestamp, the file content hash code, or its access position in a sequential list ordered by timestamp).

While we associate file objects to both version concepts (trial version and file object version), we associate logical objects only to trial versions because they are unique for each trial and already contain all execution steps (i.e., each function activations, each variable state, etc.) within a trial. Nonetheless, restoring a trial version does not restore the logical object of that trial, as it is not a tangible object, even though it is still useful for auditing or reproducing a trial.

Figure 5.6 presents an example of this version model with two trial versions for an experiment, where the user only edited “experiment.py” and added “converter.py” before executing the second trial. Circles represent object versions, and dotted squares represent trial versions. Note that the file “warp.warp” has four file object versions in Trial 1, and those versions were written four times, and read four times. Note also that Trial 1 does not have file object versions for “converter.py”, “atlas-x.ppm”, and “atlas-x.jpg” because file object versions refer to the state of files at their access time, and Trial 1 did not access these files. Equivalently, there is

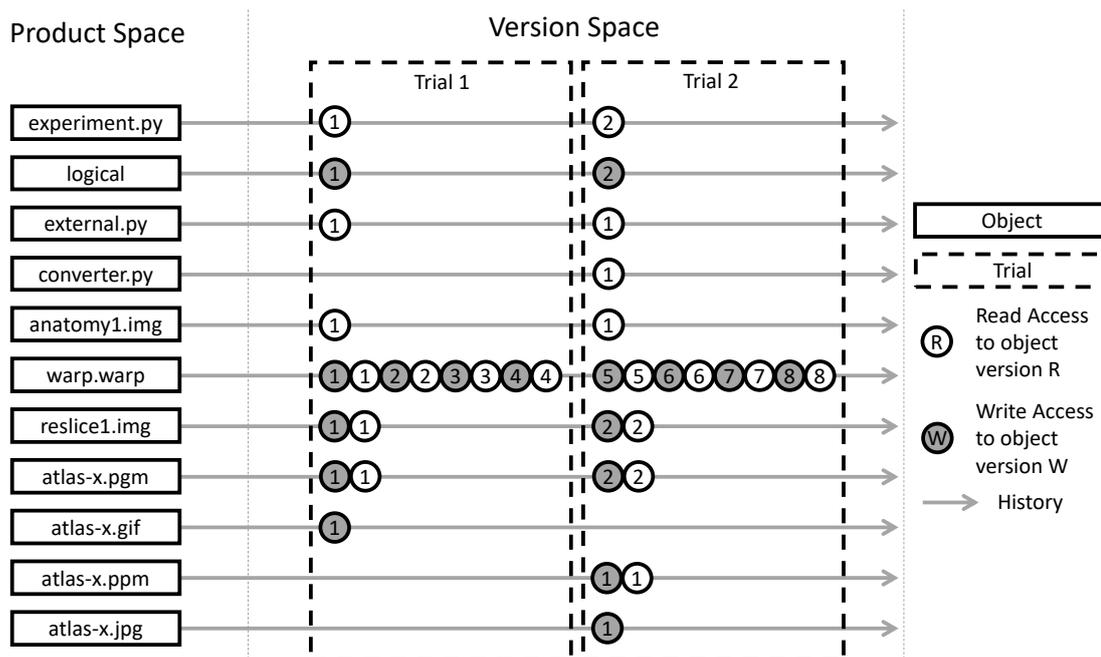


Figure 5.6: Version model example.

no file object version for “atlas-x.gif” at Trial 2, since Trial 2 did not access it.

Moreover, we can observe that both trials accessed the same file object version of “external.py” and “anatomy1.img” and that the user edited “experiment.py” after Trial 1. The logical object, on the other hand, has a single and unique version on each trial, since it contains runtime data such as function activations, start and finish times, variable values, etc. This kind of data is already time-sensitive, not demanding an extra layer of versioning. Since Trial 1 and Trial 2 have different code definitions, they have the automatic tags 1.1.1 and 2.1.1, respectively. For the remainder of this section, we will use the automatic tags to refer to the trials.

**Restore Operation.** As mentioned in Section 5.3.3, users can use trial versions to restore states of the experiment with noWorkflow 1 and 2. The main goal when restoring a trial is for reproducing it. For this reason, restoring Trial 1.1.1 would only restore the files “experiment.py”, “external.py”, and “anatomy1.img” (all at version 1). In addition, it would remove “warp.warp”, “reslice1.img”, “atlas-x.pgm”, and “atlas-x.gif”, because these files did not exist before Trial 1.1.1. However, restoring Trial 2.1.1 would restore “experiment.py” (at version 2), “external.py” (at version 1), “converter.py” (at version 1), “anatomy1.img” (at version 1), “warp.warp” (at version 4), “reslice1.img” (at version 1), and “atlas-x.pgm” (at version 1); and it would remove “atlas-x.ppm” and “atlas-x.jpg”. It would not touch “atlas-x.gif”, since Trial 2.1.1 has not accessed it. Note also that it would restore “warp.warp”, “reslice1.img”, “atlas-x.pgm” because the state of these files before Trial 2.1.1 is equal to the state after Trial 1.1.1.

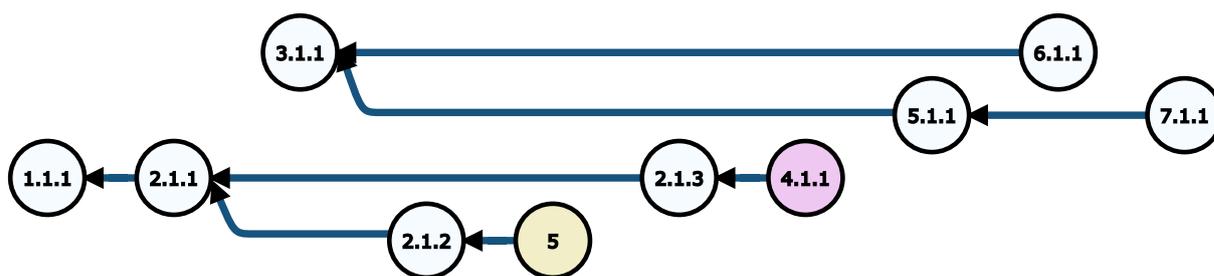


Figure 5.7: Evolution history. Nodes represent trial versions

Trial versions not only identify the state of an experiment but also track the evolution. In the example of Figure 5.6, we can see that Trial 2.1.1 is an evolution of Trial 1.1.1, because it was an execution of “experiment.py” after Trial 1.1.1. If the user executes a new script, “experiment2.py” (that is in the same directory as “experiment.py”), she would have a new trial, with version 3.1.1, but it would not be an evolution of Trial 2.1.1. However, if she executes again “experiment.py”, she would have Trial 2.1.2 based on Trial 2.1.1.

We also provide a special type of trial version to avoid losses on the restore operation. If a user changes the content of “experiment.py”, but instead of running a new trial using the modified script, she restores Trial 2.1.1, she would lose all changes. To avoid these losses, we create a special “backup” trial with the current content of all file objects in the last version (i.e., file objects edited after Trial 2.1.2). In this case, we would have the fifth trial as a backup trial, with contents of “experiment.py”, “external.py”, “converter.py”, “anatomy1.img”, “warp.warp”, “reslice1.img”, “atlas-x.pgm”, “atras-x.ppm”, and “atlas-x.jpg”. At least one of these files should be different from the ones of Trial 2.1.2 for the backup trial to be created. Note that we do not have an automatic tag for backup trials. Hence, we refer to it as Trial 5, using its id.

After restoring Trial 2.1.1, if a user runs Trial 2.1.3, it would be based on Trial 2.1.1. We keep track of this information by storing the base version of each trial. Before Trial 2.1.3, we had the base version restored to 2.1.1. After running Trial 2.1.3, we update the base version to 2.1.3. This allows our version model to track the evolution in a non-linear way. In fact, by considering the evolution of “experiment.py”, as presented in Figure 5.7, it is possible to see two branches of Trial 2.1.1: one that goes from Trial 2.1.1 to Trial 2.1.2, and another that goes from Trial 2.1.1 to Trial 2.1.3. A branch is a sequence of trials that were executed in parallel to other sequences of trials. Branches can have either a common ancestor to another branch or no ancestor at all. In this case, Trial 2.1.1 is the common ancestor of both branches, and Trial 2.1.2 and 5 belong to the same branch.

**Evolution History.** Figure 5.7 presents an evolution history longer than what we described so far. In the figure, Trials 1.1.1, 2.1.1, 2.1.2, 2.1.3, 5, and 4.1.1 are related to “experiment.py”, and Trials 3.1.1, 5.1.1, 6.1.1, and 7.1.1 are related to “experiment2.py”. We represent trials that did not finish (i.e., halted due to an error) as red nodes and backup trials as yellow nodes. According to the figure, Trial 4.1.1 did not finish, and Trial 5 is a backup trial. In addition, after getting an error on “experiment.py” execution (i.e., Trial 4.1.1), the user executed “experiment2.py” (Trial 5.1.1). Then she restored Trial 3.1.1 and executed “experiment2.py” again, creating a new branch for Trial 6.1.1. Finally, she restored Trial 5.1.1 and executed “experiment2.py”, generating Trial 7.1.1.

Note that we have two branches of “experiment.py” and two branches of “experiment2.py” in the end. Users can use branches to try different processes for their experiments and execute their experiment on the same code base, but with different input files or parameters.

**Evaluation.** Appendix B presents a short evaluation of the version model using it to answer questions from the first Provenance Challenge and the ProvBench workshops.

### 5.3.5 Summary

Table 5.2 compares provenance management in noWorkflow versions (in the bottom) with the approaches we obtained in Chapter 4. Compared to noWorkflow 0, noWorkflow 1 adds a web tool that allows *remote* access of the provenance and a version model that allows the identification of the *intention* of the evolution.

Besides noWorkflow, only eight approaches identify the intention of evolution. However, six of them (adapr, Lancet, Magni, pypet, Sumatra, and versuchung) do so by relying on Git as their version control system. These approaches have the burden of requiring users to learn how to use Git. Variolite and Verdant define version models that track the intention of variants in scripts and notebooks. Unlike these approaches that focus mostly on script definitions, noWorkflow focuses on tracking the provenance evolution. Hence, we collect not only the definition provenance but also the execution and the deployment provenance.

Although not represented in the table, noWorkflow 1 also adds an option to store the provenance in a Git repository for space efficiency and a command for restoring previous trials. noWorkflow 2 does not add many features in comparison to noWorkflow 1, but it changes the data model to link both the execution and the deployment provenance to the definition provenance.

Table 5.2: Provenance management classification.

Approach	Artifacts	Storage			Share		Versioning
		Database	Memory	File	Local	Remote	
adapr	Log, Repository, VCS	✗	✗	✓	✓	✗	Intention
Albireo	Memory	✗	✓	✗	✗	✗	✗
Astro-WISE	Oracle	✓	✗	✗	✗	✗	Sequence
Becker and Chambers (1988)	Proprietary, Source	✗	✗	✓	✓	✗	✗
Bochner, Gude, and Schreiber (2008)	PReServ	✗	✗	✗	✗	✓	✗
Chapman et al. (2021)	MongoDB	✓	✗	✗	✓	✗	✗
CPL	MySQL, PostgreSQL, 4store	✓	✗	✗	✓	✗	Trial ID
CXXR	Memory	✗	✓	✗	✗	✗	✗
Dataflow Notebook	Memory	✗	✓	✗	✗	✗	✗
Datatrack	VCS, Proprietary (CSV)	✗	✗	✓	✓	✓	Trial ID
DFAnalyzer	MonetDB	✓	✗	✗	✓	✓	✗
ES3	XML Server, GraphML, Graphviz	✓	✗	✗	✓	✗	✗
ESSW	MySQL, Content DB, Graphviz	✓	✗	✓	✓	✗	Trial ID
Flowgraph	GraphML	✗	✗	✓	✓	✗	✗
IncPy	Content DB	✗	✗	✓	✓	✗	✗
JuNEAU	PostgreSQL, Neo4j, Minio, S3	✓	✗	✗	✗	✗	✗
Lancet	Log, VCS	✗	✗	✓	✓	✗	Intention
Magni	Proprietary (JSON, HDF5), VCS	✗	✗	✓	✓	✗	Intention
Michaelides et al. (2016)	Proprietary (INPWR), PROV, Source	✗	✗	✓	✓	✗	✗
ModelKB	Content DB, Proprietary (JSON, H5)	✗	✗	✓	✓	✓	Sequence
nbgather	Memory, Notebook files	✗	✓	✗	✓	✗	Sequence
Osiris	Source	✗	✗	✓	✓	✗	✗
ProvBook	Memory, PROV	✗	✓	✗	✓	✗	✗
Provenance Curious	SQLite, GraphML	✓	✗	✗	✓	✗	✗
pypet	Proprietary (HDF5), VCS	✗	✗	✓	✓	✗	Intention
RDataTracker	PROV-JSON	✗	✗	✓	✓	✗	Trial ID
RFlow	PostgreSQL, Repository	✓	✗	✓	✗	✓	✗
Sacred	MongoDB, Relational, JSON	✓	✗	✓	✓	✗	Trial ID
SMLD	SQLite	✓	✗	✗	✗	✓	Sequence
SPADE	PostgreSQL, MySQL, H2, Neo4j, Datalog, GraphViz, PROV	✓	✗	✓	✓	✓	✗
StarFlow	OPM, Proprietary (CSV)	✗	✗	✓	✓	✗	✗
Sumatra	SQLite, VCS	✓	✗	✓	✗	✓	Intention
trackr	JSON	✗	✗	✓	✓	✗	✗

Continued on next page

Approach	Artifacts	Storage			Share		Versioning
		Database	Memory	File	Local	Remote	
TRACTUS	Memory	✗	✓	✗	✗	✗	✗
Vamsa	Memory	✗	✓	✗	✗	✗	✗
Variolite	Proprietary (JSON), VCS	✗	✗	✓	✓	✗	Intention
VCR	Log, Repository	✗	✗	✓	✗	✓	✗
Verdant	VCS	✗	✗	✓	✗	✓	Intention
versuchung	Content DB, SQLite, Proprietary (Dict), VCS	✓	✗	✓	✓	✗	Intention
WISE	Graphviz, GraphML	✗	✗	✓	✓	✗	✗
Wrattler	Content DB, JSON, Blob	✗	✗	✓	✗	✗	✗
YesWorkflow	PROV, Datalog, Graphviz	✗	✗	✓	✓	✗	✗
noWorkflow 0	Content DB, SQLite, Prolog	✓	✗	✓	✓	✗	Sequence
noWorkflow 1	Content DB, SQLite, Prolog, VCS	✓	✗	✓	✓	✓	Intention
noWorkflow 2	Content DB, SQLite, Prolog, VCS	✓	✗	✓	✓	✓	Intention

## 5.4 Provenance Analysis

Provenance analysis aims at supporting the comprehension of data and processes. noWorkflow supports analyses focusing on understanding dependencies among evaluations, activations, and file accesses, comparing trials for understanding differences, and assessing trial evolution.

### 5.4.1 Query

As described in Chapter 4, provenance tools may support *generic* and *specific* query languages. noWorkflow supports query languages in both categories. To demonstrate these languages, we will use an intentionally simple query that obtains both the code hash and the return value of the show call in line 24 of Figure 5.1 in the first execution of this script (i.e., trial 1). The following examples assume noWorkflow 2 data model.

Since all versions of noWorkflow use SQLite as their relational database, it is possible to run queries using SQL. Figure 5.8 presents an SQL query that uses the noWorkflow database. Note that we join `activation` with both `evaluation` and `code_block`, and we join `evaluation` with `code_component`. The evaluation code component represents the function call that occurs in line 24, while the activation code block represents the function definition that occurs in line 15.

```

1  sqlite> SELECT b.code_hash, e.repr
2          FROM activation a, evaluation e, code_block b, code_component c
3          WHERE a.trial_id = 1
4                AND e.trial_id = 1
5                AND b.trial_id = 1
6                AND c.trial_id = 1
7                AND a.id = e.id
8                AND a.code_block_id = b.id
9                AND e.code_component_id = c.id
10               AND a.name = 'show'
11               AND c.first_char_line = 24;
12
13  61663f8aa387cc1d3fbb9958bc44a786361c3ae9|'happy_number'
```

Figure 5.8: SQL query.

```

1  ?- activation(1, E, 'show', _, B), evaluation(1, E, _, C, _, Value, _),
2      code_component(1, C, _, _, _, 24, _, _, _, _), code_block(1, B, Hash, _).
3
4  E = 53,
5  B = 73,
6  C = 119,
7  Value = 'happy_number',
8  Hash = '61663f8aa387cc1d3fbb9958bc44a786361c3ae9'.
```

Figure 5.9: Prolog query.

SQL works fine for simple queries, but it struggles with recursive queries and queries that employ transitive closures, which are common for provenance. For instance, trying to understand whether the result of `process` in line 21 influences the result of `show` in line 24 requires navigating through the dependencies of the evaluations that appear in between. For such queries, all versions of `noWorkflow` export the provenance to Prolog facts and rules, as we described in Section 5.3.2, and support using these facts and rules for queries. Figure 5.9 presents our simple sample query using Prolog. This query uses the variables `E`, `B`, and `C` for joining the facts and the variables `Value` and `Hash` to obtain the desired results.

In addition to Prolog queries based on the collected `noWorkflow` provenance, `noWorkflow 1` also integrates with `YesWorkflow` (MCPHILLIPS et al., 2015b) for Datalog queries. Different from `noWorkflow` which performs a transparent fine-grained provenance collection, `YesWorkflow` collects provenance from annotated code blocks that users must explicitly specify. Since users design these blocks with provenance in mind, their granularity may be closer to what the users expect. However, `YesWorkflow` does not collect actual dependencies and values that occur during the execution of the script. The integration between `noWorkflow 1` and `YesWorkflow` allows users to query the fine-grained `noWorkflow` provenance using the annotated code block granularity they defined using `YesWorkflow` (PIMENTEL et al., 2016a). We call this integration `YW*NW`.

```

1  $ now show 1 -a
2  [now] trial information:
3    Id: 1
4    Status: Finished
5    Inherited Id: None
6    Script: script.py
7    Code hash: de77867d4de96c80e17161a9b4da7cafdc091fb0
8    Start: 2021-01-12 17:05:26.134200
9    Finish: 2021-01-12 17:05:26.380694
10   Duration: 0:00:00.246494
11  [now] this trial has the following function activation tree:
12    1: __main__ (2021-01-12 17:05:26.306387 - 2021-01-12 17:05:26.307682)
13      Return value: <module '__main__' from 'script.py'>
14    21: process (2021-01-12 17:05:26.306707 - 2021-01-12 17:05:26.307323)
15      Parameters: number = 10
16      Return value: 1
17    9: str (2021-01-12 17:05:26.306861 - 2021-01-12 17:05:26.306886)
18      Return value: '10'
19    11: int (2021-01-12 17:05:26.307053 - 2021-01-12 17:05:26.307076)
20      Return value: 1
21    11: int (2021-01-12 17:05:26.307162 - 2021-01-12 17:05:26.307184)
22      Return value: 0
23    24: print (2021-01-12 17:05:26.307418 - 2021-01-12 17:05:26.307652)
24      Return value: None
25    24: show (2021-01-12 17:05:26.307427 - 2021-01-12 17:05:26.307573)
26      Parameters: number = 7
27      Return value: 'happy_number'

```

Figure 5.10: Command that shows the activations of Trial 1.

As *specific query language*, all versions of noWorkflow have commands for showing the provenance data from the database. These commands are not nearly as complete as the alternatives, but they may suffice to answer simple questions. Figure 5.10 shows a command that shows the activations of trial 1, indicating their line numbers, duration, parameters, and return value in a tree format. From the results, it is possible to partially answer our original query, as it does not return the code hash of the function. Nonetheless, users can also obtain the code hash by running the command `now show 1 -d`.

noWorkflow 0 interacts with the relational database using plain SQL with a simple connection that does not support parallelism. For noWorkflow 1 and 2, we adopted SQLAlchemy as ORM (Object Relational Mapper) to have a more robust connection with support to Python objects. Consequently, it enabled a new type of specific query using the ORM objects derived from the relational tables. Figure 5.11 shows our example query using the ORM features from SQLAlchemy. At first glance, the querying logic is not much different from SQL, and it suffers from the same problems of recursive queries. However, after obtaining the mapped objects, it has the advantage of supporting helper methods and accessors that simplify the navigation to adjacent models and implement transitive closures. Note in line 20 of Figure 5.11 that it uses attributes to obtain the code block and evaluation from the activation instead of including the projection in the query.

```

1 >>> from noworkflow.now.persistence import persistence_config, relational
2 ... from noworkflow.now.persistence.models import Activation, Evaluation, CodeComponent
3 ... persistence_config.connect_existing('.')
4 ... activation = (
5 ...     relational.session.query(Activation.m)
6 ...     .join(Evaluation.m,
7 ...           (Activation.m.trial_id == Evaluation.m.trial_id)
8 ...           & (Activation.m.id == Evaluation.m.id)
9 ...     )
10 ...    .join(CodeComponent.m,
11 ...           (Evaluation.m.trial_id == CodeComponent.m.trial_id)
12 ...           & (Evaluation.m.code_component_id == CodeComponent.m.id)
13 ...    )
14 ...    .filter(
15 ...        (Activation.m.trial_id == 1)
16 ...        & (Activation.m.name == 'show')
17 ...        & (CodeComponent.m.first_char_line == '24')
18 ...    )
19 ... ).first()
20 ... print(activation.code_block.code_hash, activation.this_evaluation[0].repr)
21
22 61663f8aa387ccd3fbb9598bc44a786361c3ae9 'happy_number'

```

Figure 5.11: ORM query.

```

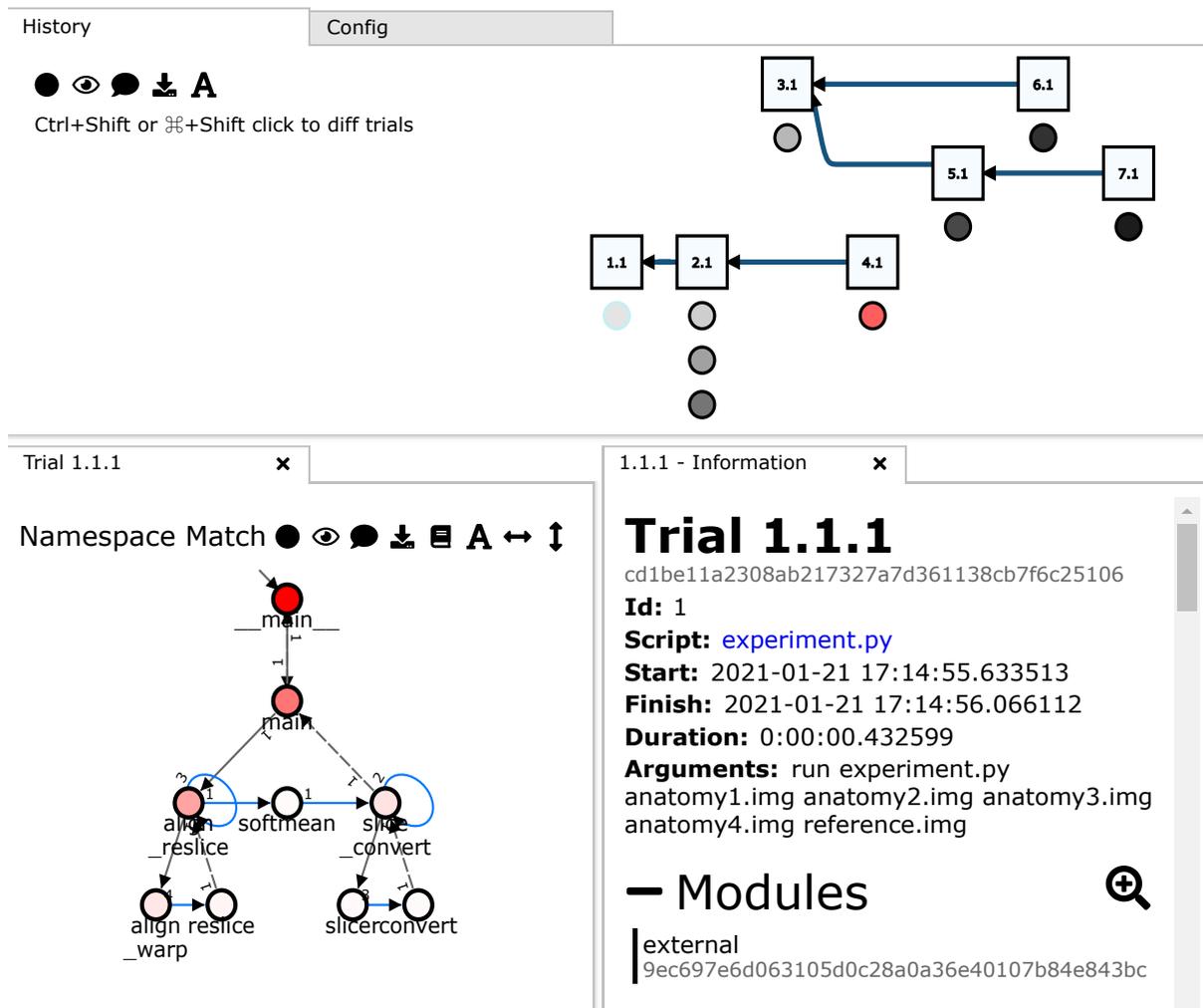
1 >>> from noworkflow.patterns import activation, code_block, code_component, evaluation, var
2 ... query = (
3 ...     activation(1, var.E, 'show', code_block_id=var.B)
4 ...     & evaluation(1, var.E, code_component_id=var.C, repr=var.Value)
5 ...     & code_component(1, var.C, first_char_line=24) & code_block(1, var.B, var.Hash)
6 ... )
7 ... for _, binds in query:
8 ...     print(binds)
9
10 {Hash: '61663f8aa387ccd3fbb9598bc44a786361c3ae9', C: 119,
11  Value: "'happy_number'", E: 53, B: 73}

```

Figure 5.12: Python pattern matching query.

For noWorkflow 2, we added an additional type of specific query language that combines the simplicity of pattern matching from Prolog queries with the power of helper methods and accessors from the ORM queries. Figure 5.12 presents our example query using this domain-specific language built on top of Python. Note that the query logic is very similar to Prolog, with similar variables having similar purposes. Still, it also exposes the ORM results to users allowing them to use attributes and methods.

In comparison to Prolog, our pattern matching also has the advantage of supporting both positional and named arguments, the advantage of not requiring the installation of external tools for querying, and the advantage of being faster since it does not require exporting the whole provenance. However, it has the disadvantage of using a harder procedural way to implement rules, instead of the declarative one in Prolog. Nonetheless, we have python implementations for all rules that we export to Prolog.

Figure 5.13: `now vis` web page.

## 5.4.2 Visualization

`noWorkflow` has many visualization features. The `now show` command output we depicted in Figure 5.10 presents an *internal log* visualization of a *complete* activation tree. `noWorkflow` 1 and 2 also display the activation tree as a visual *process graph* in the `now vis` web server we briefly described in Section 5.3.2. This web server not only displays activation trees, but also displays trial evolution graphs (such as the one in Figure 5.7), basic trial information (e.g., name, duration, arguments), environment variables, and file accesses. The web server also supports *filtering* the trials in the evolution history graph by script name or status (i.e., finished trial, unfinished trial, or backup trial). It supports *summarizing* both the evolution history graph into a summarized history and the activation tree into an activation graph. Figure 5.13 shows a screenshot of `now vis` with a summarized history graph on top, a summarized activation graph on the bottom left, and the trial information on the bottom right.

**Summarized history.** The summarized history graph in Figure 5.13 represents the same history as the one we presented in Figure 5.7. For this summarization, we use the two levels of the automatic tags we described in Section 5.3.4 to combine trials into groups. Since the summarization uses two levels, all trials in a group have the same code and the same input. That is, the summarization groups re-executions of the trial. The actual trials appear vertically as circles in the graph. The brightness of the node indicates the moment of execution: bright nodes represent old trials while dark nodes represent recent ones. This graph also uses colors to indicate the status of the trial: gray nodes indicate successful executions, and red nodes indicate failures. Note that the summarization hides backup trials, as they do not have automatic tags.

**Activation graph.** As stated before, we create the summarized activation graph in Figure 5.13 based on an activation tree. In this graph, nodes represent activations (or groups of activations), and arrows indicate the execution flow. The activations' colors indicate their duration in a scale from white to red, where white represents the fastest and red represents the slowest. Note that the main program is always red, as it encompasses the whole trial duration.

In the process of generating activation graphs, we remove edges that connect children activations to their parent activation and replace them with three types of arrows: call arrow, sequence arrow, and return arrow. A call arrow (continuous and black) goes from a parent activation to its first child to indicate the beginning of a function. Then, sequence arrows (blue) appear among activations in the same function scope to indicate their sequence. Finally, return arrows (dashed) indicate the last child activation and the end of a parent activation.

In noWorkflow 1 and 2, we support three types of activation graphs: no match, exact match, and namespace match. No match activation graphs are simple activation trees visualized as activation graphs. Exact match activation graphs combine activations in the same level (with the same parent) as long as they have the same name, in the same line, and have the same sub-structure. It allows the visualization of loops distinguishing functions that execute differently. Finally, the namespace match does not consider the sub-structure. If two functions have the same name, line, and parent, it combines them and their children. noWorkflow 0 had only the namespace variant generated by an external tool that was removed in favor of `now vis`.

**Dataflow graph.** In addition to activation graphs representing a *process graph*, noWorkflow 1 and 2 also export GraphViz graph formats with graphs that *combine* processes and data using the command `now dataflow`. Figure 5.14 presents a dataflow graph that represents a trial of Figure 5.1. In this figure, ellipses represent data (evaluations), and rectangles represent processes (activations). A rectangle may appear as a solid node, when we do not show its internal

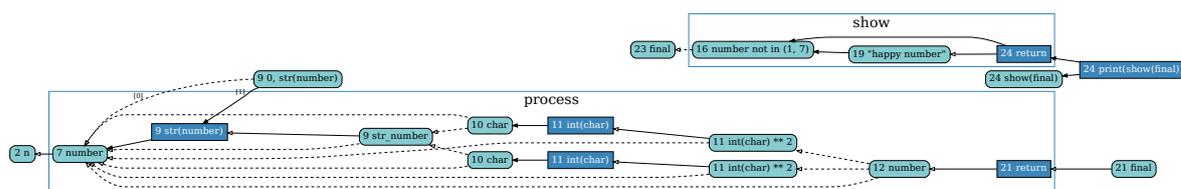


Figure 5.14: Dataflow graph.

definition or as a clear rectangle cluster with a solid `return` node when other activations occur inside of it.

Note in Figure 5.14 that the variable `final` in line 21 (bottom right) is not connected to `final` in line 23 (top left) used by `show`. It indicates that they are distinct variables that were created due to `DRY_RUN` being true. The `DRY_RUN` variable does not appear on the graph of Figure 5.14 because we generated it using the prospective mode that filters the dataflow to show only parameters, calls, and assignments to calls.

In addition to the prospective mode, `noWorkflow` supports three other dataflow modes with different *filterings*: dependency, simulation, and activation. The dependency mode presents all evaluations and dependencies in a single cluster, matching the relationships in the database without extra processing. The simulation mode is similar to the prospective mode, but it also shows relevant evaluations that occur in between calls. Finally, the activation mode is essentially a *process* graph that only displays activations. Unlike the `now vis` activation graphs that use simple sequences to construct the activation flow, the arrows in an activation dataflow only appear when data dependencies exist among activation. Hence, while activation graphs work with coarse-grained provenance, this command only works when the fine-grained provenance was collected, and it uses it for generating the graphs.

In addition to the mode *filtering*, dataflow graphs also support filtering by hiding specific types of nodes (e.g., hiding file accesses or hiding evaluations that represent types) and defining the maximum depth on the visualization of activations. Additionally, it supports some *clustering* by combining evaluations that represent the same variable or the same value.

### 5.4.3 Comparison

Comparing trials is important for verifying the reproducibility and understanding the differences among trials. `noWorkflow` has two features for comparing two trials: the command `now diff` that exists since `noWorkflow 0` and a visual comparison in `now vis` that we introduced for `noWorkflow 1` and `2` (PIMENTEL et al., 2016b, 2017).

```

1  $ now diff 1.1.1 2.1.1 -f --brief
2
3  [now] trial diff:
4  Start changed from 2021-01-08 19:42:49.538203 to 2021-01-08 19:42:50.103022
5  Finish changed from 2021-01-08 19:42:49.777175 to 2021-01-08 19:42:50.335118
6  Duration text changed from 0:00:00.238972 to 0:00:00.232096
7  Command changed from run experiment.py anatomy1.img anatomy2.img anatomy3.img
8  anatomy4.img reference.img
9  to run -b experiment.py anatomy1.img anatomy2.img anatomy3.img
10 anatomy4.img reference.img
11 Modules inherited from trial id changed from <None> to 1
12 Parent id changed from <None> to 1
13
14 [now] Brief file access diff
15 [Additions] | [Removals] | [Changes]
16 (rb) atlax-x.ppm | (w) atlas-x.git (new) |
17 (w) atlax-x.jpg (new) | (w) atlax-x.pgm (new) |
18 (w) atlax-x.pgm | (w) reslice1.hdr (new) |
19 (w) atlax-x.ppm (new) | (wb) warp.warp (new) |
20 (wb) warp.warp | ... |
21 ... |

```

Figure 5.15: Snippet of brief diff command.

The command `now diff` is similar to the command `now show` that we presented in Figure 5.10. It compares basic trial information, modules, environment variables, and file accesses. Hence, it has both *provenance* and *data* comparison. In noWorkflow 1 and 2, we introduced the option `-brief` to display a concise version of the file diff indicating only the existence of file additions, removals, and changes, instead of presenting all the detailed changes like the original command. Figure 5.15 presents a snippet of the diff command with the brief option.

All comparison available in the `now diff` command is also available as text in `now vis` with colors indicating additions (green) and removals (red). Additionally, `now vis` supports comparing activation graphs. Figure 5.16 presents a comparison of activation graphs from trials 1.1.1 and 2.1.1. In the graph comparison, nodes and arrows with black borders exist in both trials; nodes and arrows with red borders exist only on Trial 1.1.1; and nodes and arrows with green borders exist only on Trial 2.1.1. Note that `convert` activations exist only on Trial 1.1.1, while `pgmtoppm` and `pnmt/jpeg` activations exist only on Trial 2.1.1.

Moreover, nodes that exist in both trials show colors side-by-side to easy comparison. In this example, the `align_reslice` node was slightly faster in Trial 2.1.1. Hence, it shows a lighter red on the right side of the node. This comparison is also a *provenance* comparison. However, by hovering nodes in the resulting graph, it is possible to observe arguments and return values for comparing the *data*.

For creating this comparison, we convert the activation graphs back to summarized trees by removing all arrows and restoring edges that connect children activations with their parents. Note that in this operation, we keep the summarization of nodes according to the desired type of

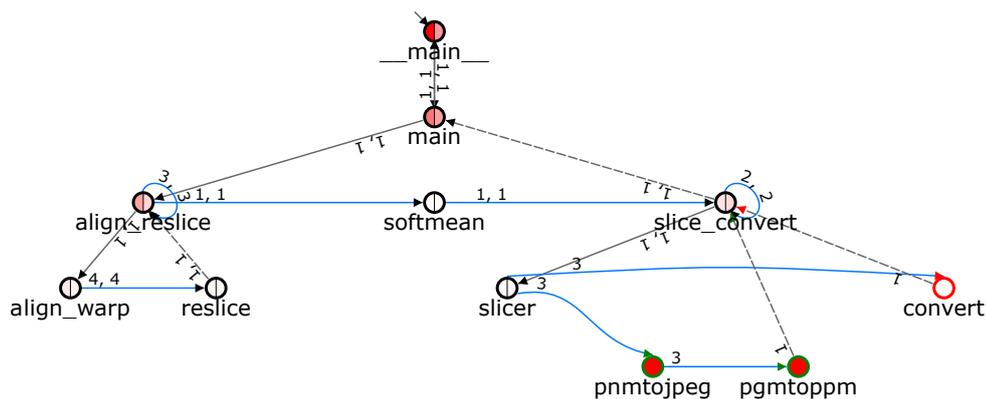


Figure 5.16: Comparison of activation graphs.

activation graph. Then, we run the tree edit distance algorithm APTED (PAWLIK; AUGSTEN, 2016) over the summarized tree to find the best mapping between the nodes in two activation trees. Finally, we restore the original arrows from both activation graphs observing whether they exist in both graphs or only in one of them.

#### 5.4.4 Summary

Table 5.3 compares provenance analysis in noWorkflow versions (in the bottom) with the approaches we obtained in Chapter 4. Compared to noWorkflow 0, noWorkflow 1 adds *functions* for querying (ORM), *combined graphs* to represent fine-grained dataflows with the ability of *filtering*, and improves the provenance comparison to include a visual activation graph diff that compares both the *provenance* and the *data*.

The integration with YesWorkflow (YW\*NW) has a different set of analyses that are based on YesWorkflow features. Hence, we present it here as a distinct row. It supports *generic* datalog queries, generates *combined graphs*, and supports filtering the graphs through datalog queries.

Finally, in comparison to noWorkflow 1, noWorkflow 2 adds pattern matching as specific queries and adds the ability to analyze the provenance externally by exporting it to Versioned-PROV. Overall, using our taxonomy as a comparison basis, noWorkflow 2 is the most diverse approach for provenance analysis. In fact, according to the taxonomy, the only feature it does not support is generating data-centric graphs.

Table 5.3: Provenance analysis classification, based on Query, Visualization, and Diff.

Approach	Query		Visualization						Diff		
			Place		Type			Sum.		Data	Provenance
			Internal	External	Log	Process	Data	Combined	Clustering		
adapr	✗	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗
Albireo	✗	Web	✓	✗	✗	✓	✗	✗	✓	✗	✗
Astro-WISE	SQL	Functions, Web	✓	✗	✗	✗	✓	✗	✗	✗	✓
Becker and Chambers (1988)	✗	Functions	✓	✗	✗	✓	✗	✗	✗	✗	✗
Bochner, Gude, and Schreiber (2008)	XQuery, XPath	Web	✗	✗	✗	✗	✗	✗	✗	✗	✗
Chapman et al. (2021)	MongoDB	PROV, Functions	✗	✓	✗	✗	✗	✗	✗	✗	✗
CPL	SPARQL, SQL	Functions	✗	✗	✗	✗	✗	✗	✗	✗	✗
CXXR	✗	Functions	✗	✗	✗	✗	✗	✗	✗	✗	✗
Dataflow Notebook	✗	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗
Datatrack	✗	✗	✓	✗	✗	✗	✓	✗	✓	✗	✗
DFAnalyzer	SQL	PROV, Web	✓	✓	✗	✗	✓	✗	✗	✓	✗
ES3	XQuery	✗	✓	✗	✗	✗	✗	✓	✗	✗	✗
ESSW	SQL	Web	✓	✗	✗	✗	✗	✓	✗	✗	✗
Flowgraph	✗	✗	✗	✓	✗	✓	✓	✓	✗	✗	✗
IncPy	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓
JuNEAU	SQL, Cypher	Web	✓	✗	✓	✗	✗	✗	✗	✗	✗
Lancet	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
Magni	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
Michaelides et al. (2016)	✗	PROV	✗	✓	✗	✗	✗	✗	✗	✗	✗
ModelKB	✗	Web	✓	✗	✓	✗	✗	✗	✗	✓	✗
nbgather	✗	Web	✓	✗	✓	✗	✗	✗	✗	✗	✓
Osiris	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
ProvBook	✗	PROV, Web	✗	✓	✗	✗	✗	✗	✗	✗	✗
Provenance Curious	SQL	Functions	✓	✗	✗	✗	✗	✓	✓	✗	✗
pypet	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
RDataTracker	✗	DDG, PROV, Functions	✗	✓	✗	✗	✗	✓	✓	✓	✓
RFlow	SQL	Web	✓	✗	✓	✗	✗	✗	✗	✗	✗
Sacred	SQL	Web	✓	✓	✓	✗	✗	✗	✗	✗	✗
SMLD	SQL	Web	✓	✗	✗	✓	✗	✗	✗	✓	✗
SPADE	SQL, Cypher, Datalog	PROV, Functions	✓	✓	✗	✗	✗	✓	✓	✓	✓

Continued on next page

Approach	Query		Visualization						Diff			
	Generic	Specific	Place		Type				Sum.			
			Internal	External	Log	Process	Data	Combined	Clustering	Filtering	Data	Provenance
StarFlow	✗	Functions, OPM	✗	✓	✗	✗	✗	✗	✗	✗	✓	✓
Sumatra	SQL	Command, Web	✓	✗	✓	✗	✗	✗	✗	✗	✓	✓
trackr	✗	Functions	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗
TRACTUS	✗	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗
Vamsa	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗
Variolite	✗	Command	✓	✗	✓	✗	✗	✗	✗	✗	✓	✗
VCR	✗	Web	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗
Verdant	✗	Web	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗
versuchung	SQL	Functions	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
WISE	✗	✗	✓	✗	✗	✓	✗	✗	✓	✗	✗	✗
Wrattler	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
YesWorkflow	Datalog	PROV	✓	✓	✗	✓	✓	✓	✗	✗	✗	✗
noWorkflow 0	SQL, Prolog	Commands, Web	✓	✗	✓	✓	✗	✗	✓	✗	✗	✓
noWorkflow 1	SQL, Prolog	Commands, Functions, Web	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓
YW*NW	Datalog	✗	✓	✗	✗	✗	✗	✓	✗	✓	✗	✗
noWorkflow 2	SQL, Prolog	Commands, Functions, Patterns, Web	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓

## 5.5 Limitations

**Collection.** Provenance collection in noWorkflow has several limitations. Regarding the definition provenance collection, noWorkflow collects input files as a whole. It cannot indicate the exact part of a file object that the trial used for deriving a given result in an experiment. For surpassing this limitation, it would be necessary to override file accessing operations to integrate it better with Versioned-PROV.

Concerning the deployment provenance collection, noWorkflow 2 still collects a snapshot of all environment variables despite moving towards a continuous approach for collecting modules. It has the drawbacks of collecting unnecessary environment variables that are never used and not accounting for changes in the variables. On the other side, the continuous approach

that `noWorkflow 2` employs for collecting modules has the possible drawback of not collecting all the necessary modules for creating variations of the experiment, since it only collects the modules that are used by a given trial – as opposed to all reachable modules collected by `modulefinder`.

Considering the execution provenance, `noWorkflow` gives few controls for users to indicate the points of the code that interest them. Users can only choose the granularity (i.e., coarse-grained activations or fine-grained evaluations), the context of the collection (i.e., main script, local modules, or all Python modules), and the depth of the collections (i.e., how many activations in the stack should the collection consider). This limitation was added by design, with the consideration that users can indicate which parts of the code interest them during the analysis. Nonetheless, it imposes a high collecting overhead that may impair the usage of `noWorkflow` in experiments that implement complex operations in pure Python. A middle-term solution to this problem could be extending `noWorkflow` with annotations to indicate which elements should be stored but keeping the collection and provenance inference at fine-grain for proper derivations.

In addition to this limitation, the fine-grained provenance collection was not designed for handling black-box operations on mutable data structures. Hence, `noWorkflow` does not use the Versioned-PROV modeling for built-in black-box operations, such as the very common `list.append` function. It is not a big issue when the added elements are accessed later – using the square brackets syntax – but it can cause problems to identify the moment of the generation of members. A solution to this problem could be allowing the definition of provenance rules for black-box operations. For instance, creating a rule indicating that `list.append` adds a member evaluation as the last member of a list evaluation.

**Management.** Provenance management in `noWorkflow` has limitations as well. The first limitation is related to the fine-grained provenance collection of complex data structures. By default, `noWorkflow` collects a representation of the values of all evaluations. Hence, when executing expressions such as `a[1]`, `noWorkflow 2` collects a representation of the values of `a`, `1`, and `a[1]`. If the representation of `a` contains all of its internal elements, collecting it defeats the purpose of Versioned-PROV in efficiently representing operations on mutable data structures. This problem can be reduced by disabling the collection of values or collecting memory addresses instead of values. Nonetheless, the default behavior is a limitation, and `noWorkflow` should be extended to only collect the full representation of mutable data structures upon their first definition or usage.

In addition to this storage limitation, `noWorkflow` has a limitation with the storage of huge

files in the content database. The old content database stores complete copies of the files, even after small changes, which may not be suitable for big files that grow over time. The new Git content database may reduce this problem by compressing and storing packfiles with deltas of the changes. Nonetheless, Git was designed for working with source code files and may not behave well with huge files.

Regarding provenance sharing and versioning, noWorkflow currently does not support merging the provenance history of multiple collaborators. If two people work at the same experiment and evolve the provenance in a different direction, they will have separate histories. Some work is underway to support the collaboration of scientists in experiments using noWorkflow.

Finally, concerning reproducibility, noWorkflow only restores the local files of experiments. It does not restore libraries to avoid breaking the installation nor environment variables. The lack of these components may prevent the reproduction of experiments. Nonetheless, noWorkflow allows the identification of the differences in these components in the case of failures. In addition to these components, noWorkflow does not restore nor collect external tools. Since it does not collect these tools, it may not help in identifying reproducibility problems when the tools are the cause.

**Analysis.** The main limitation of noWorkflow analyses is that most of them were designed for post-mortem analysis. The Prolog queries and some Python query functions require exporting the provenance to Prolog facts or in-memory graphs before querying. Even when they can be used during the execution of trials, the continuation of the execution does not allow for stream updates of these representations. Hence, it is necessary to generate everything from scratch again. The Python functions that use in-memory graphs also suffer from scalability problems should the relationship among evaluations create a huge graph.

The same issues occur for the generation of dataflow graphs and, to some extent, for activation graphs. In both cases, noWorkflow generates the graphs in memory before creating the displayable format. The problem occurs less often with activation graphs because these graphs are smaller and because noWorkflow employs techniques for reducing the issue, such as caching the graphs in the database and using the interactivity of `now vis` for accessing graph information on demand. Similar approaches could be designed for dataflow graphs.

## 5.6 Discussion

This chapter introduced noWorkflow, a tool that collects definition, deployment, and execution provenance from Python scripts transparently. We presented noWorkflow 0, the version proposed by Murta et al. (2014) and contrasted it with noWorkflow 1 and 2 according to the taxonomy we proposed in Chapter 4. noWorkflow 1 and 2 are both contributions of this thesis that use different techniques for provenance collection.

While we did not present evaluations of noWorkflow features in this chapter, parts of it have been evaluated in other work (PIMENTEL et al., 2016b, 2018b; PONTES, 2018; HU et al., 2020) and in Appendixes A and B, and Annex A. In a study about provenance evolution (PIMENTEL et al., 2016b), we evaluated how the proposed version model for noWorkflow 1 and 2 answer provenance questions from the first Provenance Challenge <sup>6</sup> and ProvBench workshops <sup>7</sup>. When we proposed the Versioned-PROV representation (PIMENTEL et al., 2018b) (i.e., the one we adopt to represent complex data structures in noWorkflow 2), we compared its space overhead with the space overhead of PROV and PROV-Dictionary (see Appendix A). Pontes (2018) evaluated the performance of using Git in noWorkflow 1 as the content database storage both in terms of space reduction and time penalty (see Annex A). We also used the proposed trial version model to answer provenance questions (see Appendix B). Finally, Hu et al. (2020) evaluated the time penalty of collecting provenance with noWorkflow 1.

In addition to these external evaluations, noWorkflow 1 and 2 have also been successfully used in other research (HU et al., 2020; LINHARES et al., 2019) and adopted by the Cloud of Reproducible Records (CoRR)<sup>8</sup>. Hu et al. (2020) uses noWorkflow 1 to extract dependencies between inputs, variables, function calls, and outputs and uses this information for identifying parts of scripts that must be re-executed after a modification for updating results. Linhares et al. (2019) uses the provenance collected by noWorkflow 2 to enrich algorithmic debugging execution trees and reduce the number of questions to users that are necessary to find a bug. CoRR is an infrastructure designed by the National Institute of Standards and Technology (NIST) for storing and distributing reproducible atoms created by different tools, including noWorkflow 2.

In this chapter we considered only the usage of noWorkflow with Python scripts. However, noWorkflow 1 and 2 also have integrations to IPython and Jupyter for provenance collection in notebooks. We discuss these integrations in the next chapter.

---

<sup>6</sup><https://openprovenance.org/provenance-challenge/FirstProvenanceChallenge.html>

<sup>7</sup><https://sites.google.com/site/provbench/home/provbench-provenance-week-2014>

<sup>8</sup><https://corr.nist.gov/>

# Chapter 6

## Provenance in Notebooks

### 6.1 Introduction

In Chapter 3, we found evidence of good and bad practices in notebooks. Among the bad practices, many notebooks have characteristics that hinder their reasoning and reproducibility, such as out-of-order cells, non-executed cells, and the possibility of hidden states. It may occur because notebooks are new tools in comparison to standard scripts and general-purpose programming languages. Hence, they lack guidelines and tools to support their development.

Despite the lack of guidelines, these bad practices are essentially related to provenance. Provenance can identify the correct cell execution order and the hidden states' presence. Additionally, provenance allows new operations in notebooks, such as cleaning uninteresting cells, extracting scripts as modules, and properly reproducing notebooks.

This chapter is organized as follows. Section 6.2 proposes a set of best practices for notebooks. Section 6.3 integrates the provenance collection and analysis of noWorkflow (Chapter 5) into Jupyter. Section 6.4 proposes a linting tool that identifies potential problems in notebooks according to the best practices and suggests fixes that aim to improve the notebook quality and reproducibility.

This chapter contains the best practices we proposed in the paper we published in the International Conference on Mining Software Repositories (PIMENTEL et al., 2019b), the IPython extension we proposed in the Workshop on the Theory and Practice of Provenance (PIMENTEL et al., 2015), and the linting tool we proposed in the paper accepted for publication in the Empirical Software Engineering (PIMENTEL et al., 2021).

## 6.2 Best Practices

In Chapter 3, we identified a set of bad practices that hinder the reproducibility and the benefits of the literate programming aspects of notebooks. Based on our findings, we propose the following best practices for the development of notebooks (PIMENTEL et al., 2019b):

- 1. Use short titles with a restrict charset (A-Z a-z 0-9 . \_ -) for notebook files and markdown headings for more detailed ones in the body.** Some operating systems may not support characters that many notebook titles use. Since notebooks support markdown, we recommend using it to write the complex titles inside the notebooks and leave the notebook title as simple as possible.
- 2. Pay attention to the bottom of the notebook. Check whether it can benefit from descriptive markdown cells. Additionally, check whether the bottom cells have been executed. If not, consider either executing or removing them.** Users seem to pay more attention to the beginning of the notebook (PIMENTEL et al., 2019b). Particularly, the bottom of notebooks usually has fewer markdown cells and fewer executed code cells, compromising reproducibility.
- 3. Abstract code into functions, classes, and modules, and test them.** Most users do not extract code into modules (PIMENTEL et al., 2019b), hindering the notebooks' reuse and test. This is especially serious because notebooks are not packed together with tests. Thus, we recommend to abstract and test notebooks.
- 4. Declare the dependencies in requirement files and pin the versions of all packages.** In Section 3.4.3, we identified that *requirements.txt* files fail less than other formats. We also recognized that many failures occur due to the lack of module dependencies. Hence, we recommend defining the dependencies explicitly and pinning the versions on a *requirements.txt* file or *Pipfile*.
- 5. Use a clean environment for testing the dependencies to check if all of them are declared.** In the reproducibility study (Chapter 3), we identified that installing dependencies in a clean environment failed more due to ImportError than just using an anaconda environment or a bloated environment. Thus, we recommend setting a clean environment and testing the notebooks dependencies before releasing them to check whether all of them are declared.
- 6. Put imports at the beginning of notebooks.** This practice is close to the PEP 8 (ROSSUM; WARSAW; COGHLAN, 2001) recommendation and helps in the verification of imports that we discussed above.
- 7. Use relative paths for accessing data in the repository.** We identified that accessing files

was also a common cause of errors in Section 3.4.3. Accessing project files using relative paths can reduce this issue.

**8. Re-run notebooks top to bottom before committing.** As presented in Section 3.4.2, many notebooks have out-of-order cells and skips. Moreover, these issues seem to impact the reproducibility (Section 3.4.3). Thus, we recommend re-running notebooks for restoring the execution counters and minimizing the impact of hidden states and out-of-order cells.

## 6.3 noWorkflow for Notebooks

In Chapter 5, we introduced noWorkflow, a tool that transparently collects provenance from Python scripts and provides mechanisms that allow users to explore this information. This section presents two integrations of noWorkflow and IPython to allow scientists to collect and analyze provenance from code executed inside Jupyter notebooks.

The first integration is an IPython extension that we propose for noWorkflow 1 and 2 that supports collecting provenance from individual cells, visualizing the collected provenance as graphs, and querying the provenance using SQL, Prolog, Python, and pattern matching (PI-MENTEL et al., 2015). We present these features in Section 6.3.1 and show that notebooks are powerful tools for interactively exploring provenance.

The second integration is a Jupyter kernel for noWorkflow 2 that transparently collects all cells' provenance in a notebook and supports using the provenance to export a clean notebook without out-of-order cells, hidden-states, and non-executed cells. We present the kernel in Section 6.3.2.

### 6.3.1 Extension

The noWorkflow extension for notebooks is composed of three parts: an IPython extension, IPython display methods in specific classes for visualizations, and a client-side extension (we use a `nbextension` for Jupyter Notebook and a similar `labextension` for Jupyter Lab). The IPython extension registers *line* and *cell magics* related to provenance collection and analysis using noWorkflow. The IPython display methods provide rich visualizations for trials, activations, history, and code blocks by outputting their data as JSON. Finally, the client-side extension reads the custom JSON formats and displays them accordingly.

### 6.3.1.1 Collection

We integrated noWorkflow's provenance collection and Jupyter Notebook by using the concepts of *line magic* and *cell magic* of IPython. While our *line magic* collects provenance from external scripts, our *cell magic* collects provenance inside notebooks.

**Line magic.** The easiest way to collect provenance from external scripts is to execute noWorkflow as is. We propose a *line magic*, `%now_run`, to perform that. One could argue that a simple shell command could perform this. However, to analyze the externally collected provenance in the notebook, a scientist would have to know the generated trial id and load a trial object that provides an interface for analysis, as we show in Section 6.3.1.2. Our *line magic* executes noWorkflow externally and returns the trial object, which can be used for immediate analysis. This *line magic* supports all arguments that the default `now_run` command supports.

**Cell magic.** While the aforementioned *line magic* improves the usability for analyzing the notebook, it is tailored to execute external scripts outside the notebook. This action would require the script to be previously created and saved into a file before running it in the notebook. To avoid this step, we propose a *cell magic*, `%%now_run`, which runs the script defined in its body. When this *cell magic* is executed, it creates a temporary file with the cell content as file content. Then, it runs noWorkflow with this file as input. Considering that the file runs externally, it is not possible to use notebook variables directly in the cell. It is only possible to pass these variables as parameters to the script. In the same way, it is not possible to directly use the trial result, but it is possible to load the output into a variable.

By default, noWorkflow uses the script name to identify trials, and we use this name in the history graphs to group trials that are probably similar. Since cells have no name, we added an optional argument (`name`) on the *magics* to indicate the trial family. With this argument, it is possible to indicate that a given trial from a specific cell belongs to a specific experiment.

**Example.** Figure 6.1 presents provenance collection using noWorkflow. The first cell loads the extension, sets the default graph width to 392px and the default graph height to 150px. The second cell uses a *line magic* to execute an external script with a custom script name (*jupext*) and returns the trial id (11). The third cell assigns a value to a variable. The fourth cell uses a *cell magic* to execute an internal script with the same name, defines that the cell output will be stored on the variable `out_var`, passes the variable `size` as argument, and returns a trial object. Note that the fourth cell result is a trial object, and it is represented as an activation graph

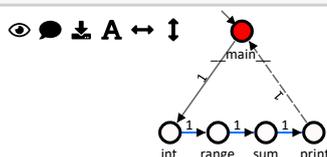
```
In [1]: %load_ext noworkflow
        %now_set_default graph.width=392 graph.height=150
```

```
In [2]: trial = %now_run experiment2.py --name jupext
        Trial.id
```

```
Out[2]: 11
```

```
In [3]: size = 5
```

```
In [4]: %%now_run --name jupext --out=out_var $size
import sys
l = range(int(sys.argv[1]))
c = sum(l)
print(c)
```



```
In [5]: out_var
```

```
Out[5]: '10\n'
```

Figure 6.1: Provenance collection in notebook using noWorkflow extension.

like the ones introduced in Chapter 5. Finally, the last cell just returns the value of `out_var`. Note that Python's `print` appends a `\n` by default.

### 6.3.1.2 Analysis

The first step in supporting provenance analysis on notebooks is to connect to the provenance database. This is accomplished by an `init` function called by `%load_ext noworkflow` that has the purpose of setting the project path on the persistence module. By default, the path is the current directory, but it is possible to call the function again to specify other directories with a named argument afterward. With access to the database, it is possible to query the provenance and use it for analysis. We propose visualizations, querying methods, and objects to perform analysis using notebooks.

**Objects.** A usual way to interact with a notebook is using its programming language for analyses. To support this use-case, we extended the SQLAlchemy ORM that maps the database tables into objects to support custom methods, caching, and specific IPython operations. For

instance, a trial object represents a single trial. It can be instantiated by specifying only the trial id or a trial tag. A trial has information about its start time, finish time, environment variables, imported modules, accessed files, code components, code blocks, activations, and evaluations. When users want to perform common queries to get the trial information and process the results using Python, they can access properties and call methods from the trial object. It will connect to the database to retrieve data ready for immediate analysis. The trial object caches some results to avoid querying the database every time. In addition, the trial object has properties and methods to retrieve other derived information, such as the trial duration.

**Visualization.** The IPython kernel supports special `_ipython_display_` methods in objects for defining how to visualize them. The default visualization of a trial object is an activation graph that shows the sequence of calls and sub-calls. An example of an activation graph is shown in *Out[4]* of Figure 6.1.

These methods support multiple formats. For instance, for dataflow graphs, we indicate that the method should display either PNG or SVG, which are formats supported by Jupyter. These methods also allow the output of custom non-supported output formats. We use custom JSON-based formats to output activation and history graphs.

For displaying custom output formats, it is necessary to implement a renderer in a client-side extension. Hence, we created both a *nbextension* for Jupyter Notebook and a *labextension* for Jupyter Lab as client-side extensions. In these extensions, we load the graphs using `now vis` modules for rich and interactive visualizations.

**Query.** As we mentioned before, visualization is not the only way to analyze provenance in noWorkflow. The trial objects have fields that can be explored. For example, the field `script_content` returns the main script content, while the field `id` returns the trial id.

It is also possible to run Prolog and SQL queries. We propose two *cell magics* to allow queries: `%%now_prolog` and `%%now_sql`. Both *cell magics* execute queries and may receive a variable result as a parameter. If they receive a variable result, the *magic* assigns the result to the variable as an iterator. If they do not receive it, the result is presented as output. The *cell magic* `%%now_sql` outputs a table where the first row is the header. The *cell magic* `%%now_prolog` outputs a list. Each entry in the list is a match. It is possible to interpolate the content of both *cell magics* with python code.

The cell magic `%%now_prolog` may also receive trial ids as parameters. The ids indicate that it should export provenance from specified trials as Prolog facts. This way, we avoid eagerly

```
[1]: import noworkflow.now.ipython as nowip
nowip.init('/home/joao/projects/demo')
```

```
[2]: trial = nowip.Trial('1.1.1')
trial.script_content[15:26]

15 def main():
16     reference = sys.argv[-1]
17     anatomy_images = sys.argv[1:-1]
18     resliced = []
19     for anatomy in anatomy_images:
20         resliced += align_reslice(anatomy,
21             atlas_image, atlas_header = softmean(*
22             for coordinate in ["x", "y", "z"]:
23                 atlas = slice_convert(atlas_image,
24
25     main()
```

```
[3]: %%now_prolog --result result 1.1.1
duration({trial.id}, slice_convert, X)
```

```
[4]: for match in result:
    print(match['X'])

0.0008490000036545098
0.0009578000026522204
0.0009693000029074028
```

```
[5]: %%now_sql
SELECT A.name AS act, F.name AS file
FROM file_access F JOIN activation A
ON A.id = F.activation_id
AND A.trial_id = F.trial_id
WHERE F.name like "%.gif"
AND A.trial_id = {trial.id}
```

act	file
convert	atlas-x.gif
convert	atlas-y.gif
convert	atlas-z.gif

```
[6]: from noworkflow.now.persistence import (
    relational, models)
from noworkflow.patterns import (
    evaluation, var)
evtab = models.Evaluation.m

def wdf(trial, last, first):
    # Prolog - find last component id
    prolog = trial.prolog.query("""
        code_name({0}, X, {1})
        """.format(trial.id, last))
    lcid = max(m['X'] for m in prolog)
    # SQL - find first component id
    sql = relational.query("""
        SELECT id FROM code_component
        WHERE trial_id={0} AND name="{1}"
        """.format(trial.id, first))
    fcid = min(r['id'] for r in sql)
    # Pattern - find last evaluation
    e = var('eid')
    pat = evaluation(trial.id, e,
        code_component_id=lcid)
    leid = max(pat, key=e.key)[0]
    # SQLAlchemy - find first evaluation
    feid = models.proxy((
        relational.session.query(evtab)
        .filter(
            (evtab.code_component_id == fcid)
            & (evtab.trial_id == trial.id)
        )
        .order_by(evtab.id)
    ).first())
    # Python method - check derivation
    return leid.was_derived_from(feid)
```

```
[7]: wdf(trial, 'resliced', 'reference')
```

```
[7]: True
```

Figure 6.2: Provenance analysis in a notebook.

loading the whole database and export facts on demand. This *magic* also loads Prolog rules that are automatically generated by noWorkflow.

**Example.** Figure 6.2 presents these possibilities of analysis. In the first cell, we imported the module `ipython` and named it `nowip`, then we called the function `init` to load the IPython magics to set the project path to `/home/joao/projects/demo`. The second cell loads a trial using the tag `1.1.1`, and presents a slice of its code content. The third cell queries the duration of `slice_convert` activations using Prolog. Note that this cell loads `trial.1.1.1` facts, interpolates the `trial.id` into the query content, and stores an iterator into the variable `result`. The fourth cell iterates through the result and prints the matches. Finally, the fifth cell performs a SQL query and outputs a table with all the activations in the desired trial that accessed files with `.gif` extension.

In addition to these simple analyses, scientists can also integrate different tools and queries since the queries' results can be obtained as Python objects and connected through Python code. The sixth cell of Figure 6.2 presents a function that receives a trial and two code component names and checks if the last evaluation of the first name argument was derived from the first evaluation of the second name argument. This function combines Prolog, SQL, pattern matching, and SQLAlchemy to obtain associated evaluation objects. Then, it uses a noWorkflow method for checking the derivation.

**Command line.** When scientists collect provenance by running noWorkflow outside a notebook, they may want to perform the analysis on a notebook due to exploratory characteristics. To ease this task, we implemented an export command-line option on noWorkflow to export notebook files related to trial objects (`now export -i`). The export command receives the trial id and generates a notebook file with the code used for loading the trial.

## 6.3.2 Kernel

While the noWorkflow extension is great for interactive provenance analyses on notebooks, using limited line and cell magics to collect provenance on cells is far from ideal. It requires an effort from the users to annotate cells and misses the provenance of the flow that connects the cells. Hence, better integration is desirable.

In Section 6.3.2.1, we propose a kernel that collects provenance from all cells transparently. Since this kernel also connects the cells with provenance, it enables new types of provenance applications. Notably, it supports cleaning the notebook, as we show in Section 6.3.2.2.

### 6.3.2.1 Collection

As explained in Chapter 3, Jupyter uses a *kernel* to execute code cells. The Jupyter interface sends a message to the kernel with the code. The kernel receives the message and executes it in the desired programming language, producing results. Then, it sends the results to the interface for display.

For designing a noWorkflow kernel, we use the IPython kernel as the foundation, which is a kernel designed to run Python code in notebooks. To collect the provenance, we override functions used by the IPython to initialize the kernel and run code cells.

Upon the kernel initialization, we start a new trial with the main code block and associated activation representing the notebook. Then, for every executed cell, we create a new code block

and associated activation for the cell, transform the cell code to collect the provenance, execute it using the original kernel run function, and save the collected provenance in the database.

Usually, activations have different namespace bindings in noWorkflow. However, since all cells in Jupyter share the same namespace, we restore the previous cell's bindings in its activation when a new cell is executed. This restoration allows creating dependencies among cells.

The IPython kernel uses the last expression in a code cell as the cell output. We keep this behavior by producing slightly different AST transformations for cells. With these transformations, we associate the cell activations with their results. In addition to displaying the last expressions, the IPython kernel stores the results into variables that other cells can access later (e.g., `Out`, `_`, `__`, `_1`, `_2`, ...). After executing a cell, we indicate that all these variables refer to the evaluation associated with the cell activation. Thus, if a posterior cell attempts to use one of these variables, we can reconstruct the dependencies.

### 6.3.2.2 Cleaning

The proposed kernel collects fine-grained provenance from notebooks, which users can analyze using all the methods presented in Chapter 5 and Section 6.3.1. In addition to these methods, the fine-grained notebook provenance supports a different type of provenance application: notebook cleaning.

As identified in Chapter 5, many notebooks have out-of-order cells, non-executed cells, and the possibility of hidden states. Notebook cleaning uses the provenance collected from a notebook to create a new notebook with the correct cell execution order, with all relevant executed cells (i.e., no hidden states), and without non-executed cells – they are not part of the provenance.

The `now clean` command in noWorkflow 2 performs this operation. Users can specify a set of evaluations to indicate the notebooks' relevant parts or consider the whole notebook as relevant. Suppose they consider the whole notebook as relevant. In this case, the cleaning operation just creates a new *history* notebook with all executed cells in order, which guarantees that the new notebook has no hidden states nor non-executed cells.

On the other hand, if the user specifies a set of evaluations as relevant, we use these evaluation's provenance to clean the history notebook. To clean the history notebook, we navigate the dependencies of the relevant evaluations to find all the evaluations that contribute to their generation. These evaluations belong to code cells that we mark for inclusion in the clean note-

book. These cells often have evaluations that derive from other cells. Hence, we mark these evaluations as relevant as well and repeat the process until there is no new relevant cell. We use the relevant cells with their order of execution to create a valid clean notebook.

While this process works for creating a clean notebook composed of code cells, it does not work to restore Markdown cells for two reasons. First, Jupyter does not send Markdown cells to the kernel. Hence, we do not have the provenance of Markdown cells. Second, Markdown cells do not contribute to the generation of any relevant evaluation. Thus, even if we had the provenance of Markdown cells, they would not appear in the final result.

To solve this problem, we have an optional extra step in the cleaning operation to restore Markdown cells in a clean notebook. For this step, the user specifies the original notebook's final version (i.e., the dirty notebook with all the Markdown cells). We use the Longest Common Subsequence (LCS) algorithm (HIRSCHBERG, 1977) to match the cells from the original notebook with the cells from the clean one.

Knowing the position of matched cells allows us to identify whether a Markdown cell appears before, after, or between matched cells. We use this information to insert Markdown cells in the proper positions of a clean notebook. In some hidden-state and out-of-order situations, the matched cells indicate ambiguous positions for Markdown. Thus, the operation supports adding the Markdown either before (by default) or after ambiguous code cell positions.

Figure 6.3 depicts the notebook cleaning based on a dirty notebook with out-of-order cells, non-executed cells, and hidden states (a). Note that this notebook is not reproducible neither by following the cell execution order nor by following the cell execution order. Figure 6.3(b) presents a history notebook that uses the provenance to reconstruct the original cell execution order with all the cells.

We selected the evaluation of `co` in cell [7] as relevant for cleaning and passed the original notebook as the argument to reconstruct the Markdown. The cleaning operation found that only cells [4]–[7] are relevant to the selection criteria. Then, the Markdown reconstruction found matches for two cells ([5], [7]) and ambiguities for the other two. In Figure 6.3(c), the reconstruction operation added the top Markdown cells before the ambiguous cell [4] and the “View” Markdown cell before the ambiguous cell [6]. In Figure 6.3(d), the reconstruction operation added the top Markdown cells after the ambiguous cell [4] and the “View” Markdown cell after the ambiguous cell [6].

Original.ipynb	now clean	now clean -n 7 -c co -j Original.ipynb	now clean -n 7 -c co -j Original.ipynb --merge-last
Set initial value [ ]: <code>co = 1</code>	[1]: <code>co = 2</code>	Set initial value	[4]: <code>co = 0</code>
[1]: <code>co = 2</code>	[2]: <code>co += 1</code>	Increment twice [4]: <code>co = 0</code>	Set initial value
Increment twice [6]: <code>co += 1</code>	[3]: <code>co</code>	[5]: <code>co += 1</code>	Increment twice
View [7]: <code>co</code>	[3]: 3	View [6]: <code>co += 1</code>	[5]: <code>co += 1</code>
[7]: 2	[4]: <code>co = 0</code>	[7]: <code>co</code>	[6]: <code>co += 1</code>
Reset [4]: <code>co = 0</code>	[5]: <code>co += 1</code>	Reset [7]: 2	View [7]: <code>co</code>
[8]: <code>co += 10</code>	[6]: <code>co += 1</code>	Finish notebook	Reset [7]: 2
Finish notebook	[7]: <code>co</code>		Finish notebook
[7]: 2	[7]: 2		
[8]: <code>co += 10</code>	[8]: <code>co += 10</code>		
(a)	(b)	(c)	(d)

Figure 6.3: Notebook cleaning using provenance.

## 6.4 Julynter

Provenance collection and notebook cleaning can solve the problem of out-of-order cells, hidden states, and non-executed cells. However, it has drawbacks as it only operates post-mortem and requires a heavy provenance collection that might affect the experiments' performance. In this section, we propose an approach that aims at minimizing these problems during the live development of notebooks, using simple provenance information that is readily available in the IPython kernel.

Based on the results of our analyses in Chapter 3 and the proposed best practices in Section 6.2, we propose Julynter<sup>1</sup>, a tool that performs linting on notebooks. Julynter is a Jupyter Lab extension that performs many checks on the quality and reproducibility of notebooks in real-time and produces recommendations.

This section is organized as follows. Section 6.4.1 describes the approach. Section 6.4.2 presents the experiment design we defined to evaluate Julynter. Section 6.4.3 indicates how we collected the experiment data. Section 6.4.4 presents the experiment results. Finally, Section 6.4.5 describes the threats to the validity of the Julynter experiment.

<sup>1</sup><https://dew-uff.github.io/julynter>

The screenshot shows the Julynter interface. The top menu includes File, Edit, View, Run, Kernel, Tabs, Settings, and Help. The notebook title is 'UNTITLED2-COPY1.IPYNB'. The code in the notebook is as follows:

```
[3]: def fib(x):
      if x <= 1:
          return x
      return fib(x-1) + fib(x-2)
      fib(10)
```

```
[3]: 55
```

```
[8]: from matplotlib import pyplot
      %matplotlib inline
      x = range(15)
      y = [fib(n) for n in x]
      pyplot.plot(x, y);
```

```
[8]: [<matplotlib.lines.Line2D at 0x7fb818874ac8>]
```

The plot shows the Fibonacci sequence for n from 0 to 14. The x-axis ranges from 0 to 14, and the y-axis ranges from 0 to 350. The curve starts at (0,0) and increases exponentially, reaching approximately 377 at n=14.

```
[3]: fib(15)
```

```
[3]: 610
```

```
[ ]:
```

The left sidebar displays linting recommendations categorized into four groups:

- Invalid Title**
  - Title starts with "Untitled"
  - Title has "-Copy"
- Hidden State**
  - Cell 3 has changed since its execution, but it wasn't executed after the changes
  - Cell 3 repeats the execution counter 3
  - Cell 3 skips the execution counter
  - Cell 2 skips the execution counter
- Confuse Notebook**
  - Cell 3 has the execution counter 3 in the wrong order
  - The last cell of the notebook is not a markdown cell
- Import**
  - Cell 2 has import but it is not the first cell
  - Module matplotlib was imported by Cell 2, but it is not in the requirements file

Figure 6.4: Julynter in action (left pane). By analyzing the notebook on the right pane, Julynter identified ten issues from four different categories.

### 6.4.1 Approach

Figure 6.4 presents Julynter in action for the notebook of Figure 3.1. Julynter recommended ten changes related to four categories: Invalid Title, Hidden State, Confuse Notebook, and Import. In addition to these categories, Julynter also has an Absolute Path category.

In addition to showing linting recommendations to users, Julynter also has filtering and display features to provide better readability. Users can filter recommendations by category, recommendation code, and appearance in a specific cell. Additionally, they can group the recommendations by category or by cell through the interface. They can store their preferences in

the notebook, the project folder (i.e., the working directory of the Jupyter Lab execution), or the user directory.

The interface also allows users to click on the recommendations to apply actions. In the Invalid Title recommendations, it opens the rename notebook form. In the Import recommendation related to an import that does not exist in the requirements file, it adds the imported package to the requirements file, indicating its version. In a recommendation related to a cell that depends on a variable that was defined in a cell that does not exist anymore, it recreates the cell. Finally, for the other recommendations, it moves to the cell with the issue to allow users to fix them.

Julynter currently identifies 21 issues from notebooks. Table 6.1 presents these issues with their categories and the Julynter recommendations on how to fix them. Note that some recommendations require a kernel restart to really ensure the reproducibility. After some feedback from user experiments, we added a button to hide these recommendations for development notebooks. The Julynter extension detection covers six out of the eight best practices proposed in Section 6.2. To cover the seventh (using a clean environment for testing dependencies), we added a command-line interface (CLI) to Julynter that allows users to use pyenv environments, Conda environments, or Docker containers to detect dependency files and install them. Users can use this CLI to check if the installation is enough to import all modules. They can also use it to check the reproducibility of the notebooks after installing the modules. Finally, they can use it to automatically prepare an isolated environment with only the project dependencies. Hence, the only best practice that Julynter still does not cover is suggesting users to abstract code. Nonetheless, this suggestion is on our radar for future releases.

For detecting the issues, Julynter has two linting modules: a language-agnostic and a language-specific one. The language-agnostic module checks for common issues on the notebook structure that do not depend on the notebook language. This is the case for issues C1, C2, C3, C4, C5, H3, H4, T1, T2, T3, T4, T5, T6, T7. The language-specific module connects to the kernel to obtain basic provenance information about the execution history, the cell dependencies, the executed cells with absolute paths, and the status of imported modules on requirement files (issues H1, H2, H5, I1, I2, P1). This provenance does not require a heavy load on the collection since the cell execution history and the variable namespace is readily available in the IPython kernel during the execution. Both modules connect to each other using Jupyter Comm<sup>2</sup>. Hence, they do not interfere with the execution.

Figure 6.5 presents the architecture of Julynter and Jupyter communications. When the

---

<sup>2</sup><https://jupyter-notebook.readthedocs.io/en/stable/comms.html>

Jupyter Lab web application sends a cell to the kernel to execute, the Julynter extension triggers both linting modules. The language-specific module sends an invocation of a query function to the kernel, which then returns the execution history, the cell dependencies, the imports, and the absolute paths. Julynter processes this data together with the notebook definition and presents it back in the Jupyter Lab Application. The language-agnostic module processes only the notebook definition to report the issues.

Julynter has some limitations. First, the detection is restricted to run as an extension of Jupyter Lab. Currently, it cannot run as a standalone module nor as a Jupyter Notebook exten-

Table 6.1: Issues detected by Julynter. The first character of the Code indicates the category: C – Confuse Notebook; H – Hidden State; I – Import; P – Path; T – Invalid Title

Code	Message	Suggestion
C1	Cell <i>:index</i> is a non-executed cell among executed ones.	Please consider cleaning it to guarantee the notebook reproducibility.
C2	Cell <i>:index</i> has the execution counter <i>:excount</i> in the wrong order.	Please consider re-running the notebook to guarantee its reproducibility.
C3	Cell <i>:index</i> is empty in the middle of the notebook.	Please consider removing it to improve the notebook readability.
C4	The first cell of the notebook is not a Markdown cell.	Please consider adding a Markdown cell to describe the notebook.
C5	The last cell of the notebook is not a Markdown cell.	Please consider adding a Markdown cell to conclude the notebook.
H1	Cell <i>:index</i> has execution results, but it was not executed in this session.	Please consider re-executing it to guarantee the reproducibility of the notebook.
H2	Cell <i>:index</i> has changed since its execution, but it was not executed after the changes.	Please consider re-executing it to guarantee the reproducibility of the notebook.
H3	Cell <i>:index</i> repeats the execution counter <i>:excount</i> .	Please consider re-running the notebook to guarantee its reproducibility.
H4	Cell <i>:index</i> skips the execution counter.	Please consider re-running the notebook to guarantee its reproducibility.
H5	Cell <i>:index</i> uses name “ <i>:variable</i> ” that was defined in <i>In [:excount]</i> , but it does not exist anymore.	Please consider restoring the cell and re-running the notebook to guarantee its reproducibility.
H6	Cell <i>:index</i> has the following undefined names: <i>:undefined</i> .	Please consider defining them to guarantee the reproducibility of the notebook.
I1	Cell <i>:index</i> has import but it is not in the first cell.	Please consider moving the import to the first cell of the notebook.
I2	Module <i>:module</i> was imported by Cell <i>:index</i> , but it is not in the requirements file.	Please consider adding it to guarantee the reproducibility of the notebook.
P1	Cell <i>:index</i> has the following absolute paths: <i>:paths</i> .	Please consider using relative paths to guarantee the reproducibility of the notebook.
T1	Title is empty.	Please consider renaming it to a meaningful name.
T2	Title starts with “Untitled”.	Please consider renaming it to a meaningful name.
T3	Title has “-Copy”.	Please consider renaming it to a meaningful name.
T4	Title has blank spaces.	Please consider removing them to support all OS.
T5	Title has special characters.	Please consider replacing them to support all OS.
T6	Title is too big.	Please consider renaming it to a shorter name and using a Markdown cell for the full name.
T7	Title is too small.	Please consider renaming it to a meaningful name.

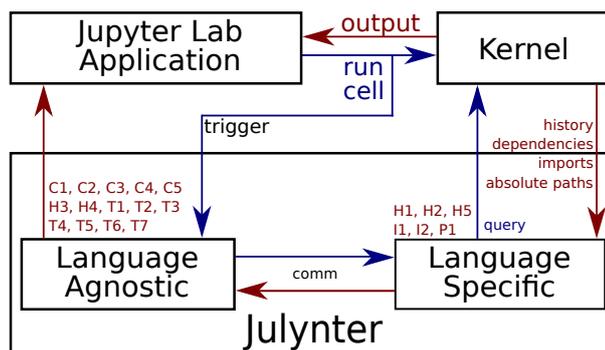


Figure 6.5: Architecture of Julynter. Blue arrows represent input messages that occur before the cell execution. Red arrows represent output messages that occur after the kernel executes the cell.

sion. Second, it must be executed in real-time. Starting Julynter in an existing notebook with a new kernel results in many warnings related to the presence of results from previous executions, and no warnings related to imports and absolute paths. This situation can be easily solved by running the whole notebook again, but it may not be what users expect when they use traditional linting tools. Finally, the language-specific module currently only supports Python.

## 6.4.2 Experiment Design

Since Julynter connects to the kernel to get the execution history in real-time, it is more capable of detecting hidden states and other issues than we were in the reproducibility study of Chapter 3. However, since this detection relies on real-time history, it cannot detect hidden states in notebooks executed in previous sections. Thus, we could not simply use the notebooks we collected in Chapter 3 to evaluate Julynter as it would at most produce the results we presented before.

Hence, for evaluating the usability and capability of Julynter to ensure the quality of notebooks in the wild, we designed an experiment with users using Julynter over their own notebooks. The experiment was composed of three parts: a characterization form, the main experiment, and an exit questionnaire.

In the characterization form, we asked questions about how frequently do they use notebooks, their experience with linting tools, Jupyter Notebook, Jupyter Lab, Python, R, and Julia, their preference between Jupyter Lab and Jupyter Notebook, and their usage of notebooks.

Due to the COVID-19 pandemic, we had to run the experiment remotely. Hence, for the main experiment, we adapted Julynter to collect usage data and asked the participants to install Julynter in their own machines and use it with their own notebooks for a week. For collecting

the usage data, we also asked the participants to run a configuration tool to indicate which data they would like to share. Additionally, we added buttons for each recommendation in the tool to allow users to send feedback through positive, negative, and textual reports.

In the exit questionnaire, we asked users to send their collected data. We also asked about their satisfaction with each linting category using a Likert scale, and their overall satisfaction with the tool using both a System Usability Scale Questionnaire (BROOKE, 1996) and Microsoft Reaction Cards (BENEDEK; MINER, 2002). Finally, we asked for suggestions to improve Julynter.

### 6.4.3 Data Collection

We conducted the experiment in three phases: I, II, and III. Phase I was a pilot and had the goal of identifying problems in the experiment itself. Two people participated in this phase: one advisor and one undergrad student, and they identified five minor problems in the experiment. We do not use their results in the next section.

After fixing the experiment problems, we directly invited ten people for the next phase of the experiment. We selected these people based on our knowledge that they use notebooks. Only six of them completed the experiment during Phase II, and all six gave feedback on how to improve the tool. We implemented the requested features and started the last phase of the experiment. We shared the experiment in Data Science groups, Graduate Student groups, Python groups, and Twitter for this phase. Two people that were invited to Phase II but did not have time for the main experiment decided to participate in Phase III. Fourteen people answered the initial form, but only six completed the experiment. Figure 6.6 presents the flow of completion of the experiment for the main phases. Note that one participant did not reply to our invitation in Phase II, and one interrupted the experiment after filling the initial form. In Phase III, eight participants interrupted after the initial form, and two interrupted after starting the experiment.

Figure 6.7 presents the experience of the 12 participants that concluded either Phase II or Phase III. While all of them have at least an average experience with Jupyter Notebook and Python, most of them are novices in Jupyter Lab, which is the tool Julynter supports. It is expected, as Jupyter Lab is a newer tool released in 2018. When we asked which tool they prefer, seven participants prefer Jupyter Notebook, four participants prefer Jupyter Lab, and a participant has never used Jupyter Lab to have a preference.

When we asked the participants to report their use-cases for Jupyter in a text field (i.e., we did not give predefined options and a participant could write multiple things), nine participants

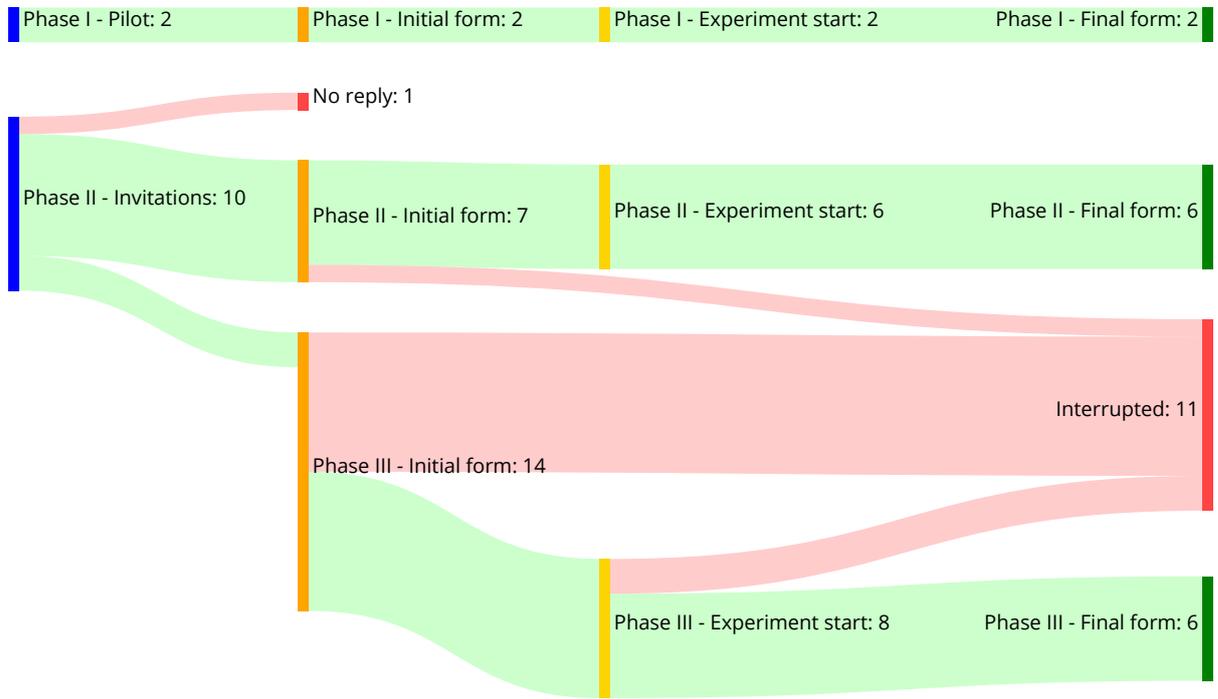


Figure 6.6: Participants experiment flow.

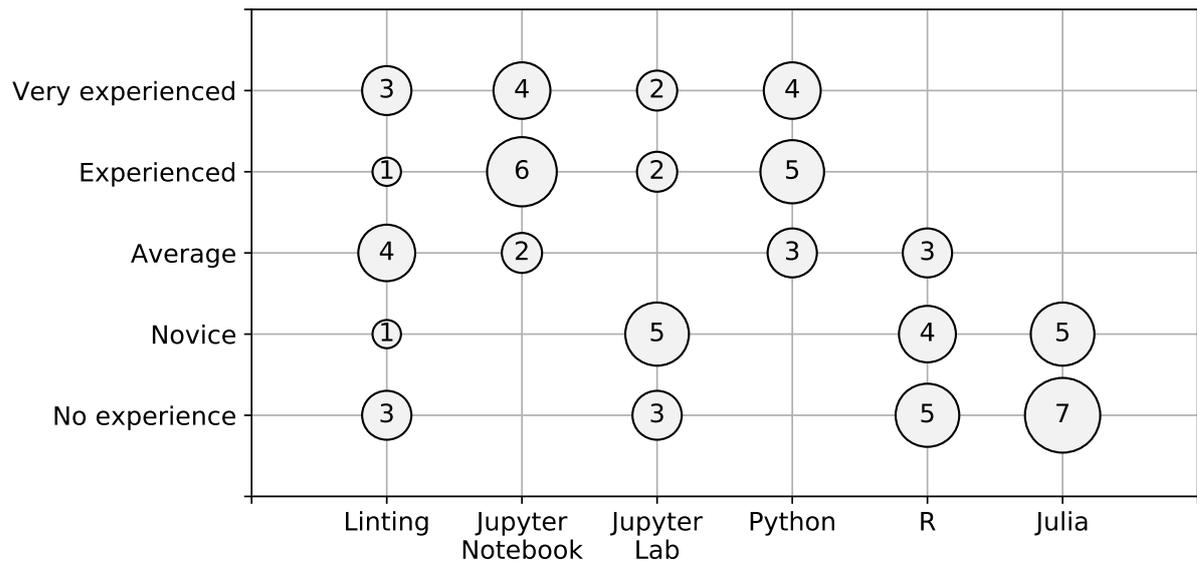


Figure 6.7: Participants' experience.

Table 6.2: Julynter usage statistics.

Phase	P#	Days	Sessions	Notebooks	Lints	Solved	Solved (%)	Lint Clicks
II	#1	4	2	1	77	72	93.5%	62
II	#2	3	2	1	330	330	100.0%	218
II	#3	4	10	4	317	217	68.5%	202
II	#4	5	8	2	201	154	76.6%	129
II	#5	2	4	1	71	48	67.6%	40
II	#6	1	1	1	587	521	88.8%	124
III	#7	6	18	15	602	534	88.7%	460
III	#8	3	7	1	1,888	1,873	99.2%	880
III	#9	3	29	7	106	66	62.3%	85
III	#10	1	22	8	85	43	50.6%	54
III	#11	5	28	4	1,053	751	71.3%	333
III	#12	2	14	4	58	20	34.5%	34
<b>Total</b>	<b>12</b>	<b>28</b>	<b>145</b>	<b>49</b>	<b>5,375</b>	<b>4,629</b>	<b>86.1%</b>	<b>2,621</b>

answered data-centric use-cases, such as data analysis, data cleaning, and data visualization; four use Jupyter for prototyping scripts and tools; four use or have used Jupyter for education tasks such as preparing course material or doing homework; three use it for research; two use it for communicating results and workflows; and one uses Jupyter to build interactive reports.

During the experiment, seven participants worked on data analysis projects, four participants used notebooks as scratchpads for prototyping and developing packages, and one participant prepared class materials.

In the next subsection, we present the experiment results, filtering out both the participants of the pilot experiment (Phase I) and the participants that did not conclude the experiment.

#### 6.4.4 Results and Discussion

**Usage.** Since the participants used Julynter at their own pace with their own notebooks, the number of recommendations they received varied. Table 6.2 presents the number of days, usage sessions, notebooks each participant worked on, and the number of lint recommendations they received, solved, or clicked. We count a usage session as the moment a participant opens a notebook in the Jupyter Lab interface. Note that while many participants worked on a single notebook during the experiment, most of them opened the same notebook multiple times and through many days.

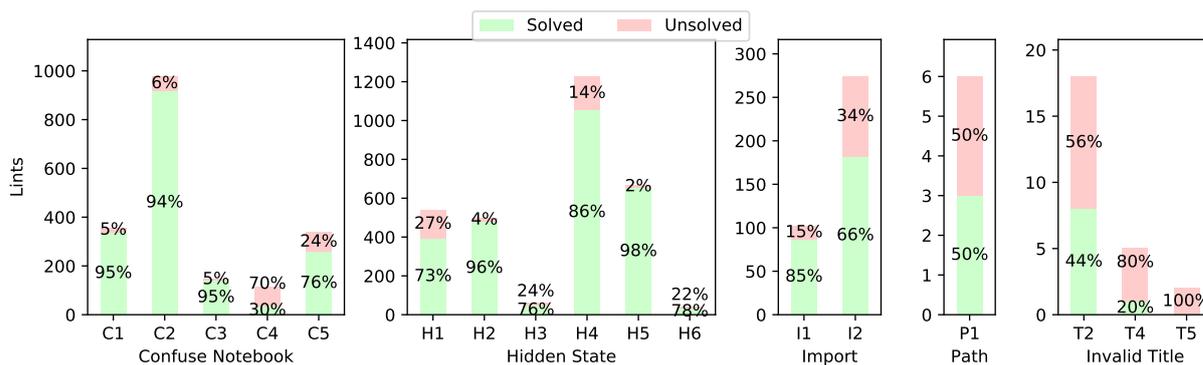


Figure 6.8: Solved and unsolved lints.

The participants #1 and #2 worked on the same usage session through different days, indicating that they did not close Jupyter Lab from a day to another. #6 was the only participant that worked on a single usage session of a single notebook during the experiment. Nonetheless, #6 was also the participant that received the most lint recommendations during Phase II. During Phase III, #10 also used Julynter for a single day, but in eight notebooks across 22 kernel sessions.

In this table, we count lints as all recommendations that Julynter shows and solved lints as all recommendations that disappear after a participant action. Despite the tool showing hundreds of recommendations to most participants, this number does not reflect directly on the effort they had to solve them. For instance, opening a big notebook with execution results leads to many H1 recommendations, indicating that it has results from previous kernel sessions. Solving them is as easy as running all notebook cells. On the other hand, solving cells with H4 recommendations (which identify skips) requires restarting the kernel and re-running all cells.

**Recommendations.** Figure 6.8 presents all lints that appeared to the participants, indicating the percentage of solved and unsolved lints. The recommendations T1 (empty title), T3 (title with “-Copy”), T6 (big title), and T7 (small title) did not appear for any participant. As expected, recommendations that can appear for any cell were more prevalent than the ones that appear for the notebook (C4, T1 – T7) or in sporadic events such as importing modules (I1 – I2) or using absolute paths (P1). Recommendations related to the organization of the notebook (H4 – skips, C2 – out-of-order cells) appeared the most.

These results suggest that Julynter recommends changes to improve the quality of the notebook that the participants are willing to apply. Nonetheless, the participants solved more some types of recommendations than others.

**Recommendation Feedback.** In the *Confuse Notebook* group, C4 and C5 were the least solved recommendations. These recommendations suggest using Markdown cells in the beginning to describe the notebook and in the end to conclude it, respectively. We received four negative reports about C5, three textual reports asking why it was necessary, and one textual report complaining that it appeared too soon (i.e., before finishing the notebook to draw conclusions). C4 was more controversial: we received two negative reports and three positive ones about it. #10 sent a textual report indicating that the recommendation was good, but it would not be fixed because the notebook was part of a tool written by someone else. #12 sent both positive and negative reports about it, with a textual report indicating that “not all notebooks are literate ones”.

In the *Hidden State* group, participants solved the least H1, H4, and H6 recommendations. As described before, H1 appears when the user first opens a notebook that has results from previous executions. If the user does not want to run the notebook, it is expected not to have it solved. We received two negative reports with textual reports. A participant indicated that the notebook was not executed yet. The other indicated that an error in a previous part of the notebook prevented its normal execution.

The recommendation H4 is harder to solve, as it requires restarting the kernel and re-running all cells. This recommendation received a textual report indicating that the participant did not understand the suggestion. It also received a positive report. Related to this recommendation, in the exit questionnaire, two participants suggested that linting notebooks should occur in two phases: a phase for supporting exploratory analyses with skips in the cell execution counter and a phase to guarantee the reproducibility.

The recommendation H6 appears when a cell uses a variable that is not defined in the notebook. A participant sent a textual report indicating that the recommendation was not appropriate because the variable was actually defined. When we analyzed the notebook code, we noted that a widget uses IPython functions to change the global state. As this is a very unusual situation, we suggest using Julynter filters for this type of false-positive recommendation.

In the *Import* group, participants solved the least the recommendation I2 (adding packages to “requirements.txt”) and two of them sent textual reports indicating that they do not use these files. Once again, #12 sent both positive and negative reports in different notebooks. The other recommendation (I1 – moving imports to the beginning) also received feedback. A participant sent a negative report without indicating why, but two participants sent positive reports. #10 sent a textual report indicating that imports should indeed stay in the first cell, but the issue would not be fixed as the notebook was designed by someone else.

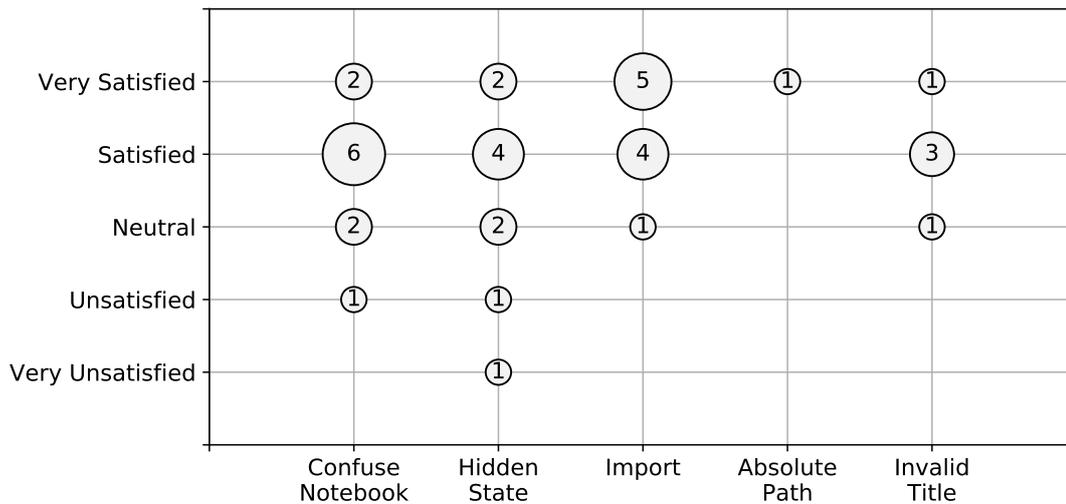


Figure 6.9: Satisfaction with the lint groups.

The recommendation to not use absolute *Paths* (P1) only appeared for two participants. One could not solve it due to a bug that Julynter had during Phase II in Windows. The participant who experienced the bug sent a positive report about the suggestion, though.

Finally, the *Invalid Title* group had the recommendations the participants solved the least: T5 (title with special characters), T4 (title with blank spaces), and T2 (title with “Untitled”). A participant sent a negative report about T4, asking why the title could not have blank spaces. The same participant sent a positive report about T2.

In the exit questionnaire, we asked the participants to indicate the extent they are satisfied with each linting recommendation group. Figure 6.9 presents the results. In this figure, we filtered out groups that did not appear for the participant. The participants are mostly satisfied with the recommendations, and the only groups that caused dissatisfaction are the *Confuse Notebook* and *Hidden States*. The participants that did not like these groups are the ones that sent negative feedback to C5, H1, and H4.

**Usability.** We used the System Usability Score (SUS) (BROOKE, 1996) to evaluate the usability of Julynter. This score is calculated based on ten standard statements presented in the exit questionnaire. The user can select answers ranging on a Likert scale from 1 (completely disagree) to 5 (completely agree). We calculated an average SUS score of 77.5 on a scale between 0 and 100. According to Bangor, Kortum, and Miller (2008), this is a good score in the acceptability range. The minimum score was 52.5, which is close to the minimum OK score in the acceptability range. On the other hand, the maximum score was 100, which is categorized as the best imaginable score (BANGOR; KORTUM; MILLER, 2008).



- Added a button to filter all recommendations that require restarting the notebook based on the feedback of two participants (#2 and #6). These participants identified that the development of notebooks occurs in two phases: (i) writing the notebook in an exploratory manner, and (ii) ensuring the quality and reproducibility. Hence, a filter that disables recommendations that require restarting aims to support the first phase.
- Redesigned the tool architecture to support other programming languages based on the request of #1 to support multiple programming languages besides Python.
- Loaded the interface font-size from Jupyter Lab configuration files based on the request of #3 to support configurable sizes.
- Started saving the recommendation results on the notebook metadata and implemented a command-line tool to check the stored recommendations based on the request of #4. This participant would like to have icons on GitHub and GitLab expressing how adherent to recommendations are the notebooks. Implementing a CLI was the first step towards this goal.
- Improved the options to filter recommendations, adding an option to filter specific ones, based on the feedback of #6.

We also contacted the participants of Phase II after the changes and asked them what they thought about the changes. #1 did not reply. #3 had an issue updating and running Julynter and also did not give feedback. The other four participants liked the changes. #4 would like to see the notebook compliance to Julynter recommendation in platforms like GitHub and GitLab, but recognizes that it is outside our scope. #6 suggested some minor interface changes, such as changing icon colors, rewording recommendations, and removing menu items. We intend to apply some of these suggestions in the future.

In Phase III, we received feedback suggesting to implement the linting as a command-line interface, suggesting to reword a recommendation, and suggesting to add an option to indicate that the notebook should not change the style. We cannot implement all Julynter recommendations in a command-line interface, since some of them depend on the execution state. However, we intend to implement the other two by changing the lint message as requested and adding a button to disable Julynter altogether.

Besides these reports, some participants of both phases expressed concerns about the data intrusiveness of the experiment itself. Some even suggested designing a controlled remote experiment in a virtual machine or Binder for security reasons. We considered it when designing

the experiment, but we anticipated it would be artificial and would not detect which recommendations occur in the wild nor whether the recommendations are good enough for users to apply in their own notebooks.

Moreover, some indicated that they use Jupyter Notebook instead of Jupyter Lab daily and did not run the experiment much. Finally, some reported bugs that were not caused by Julynter.

A participant indicated that despite all effort with linting, “the biggest problem remains the lack of training of scientists in software engineering”.

### 6.4.5 Threats to Validity

The Julynter experiment also has some threats to validity that we depict below.

**Internal.** We selected participants for Phase II based on our previous knowledge that they used Jupyter. This may bias the selection of participants to close contacts. In an attempt to mitigate this threat, we distributed the invitation for Phase III to public data science and research groups in Telegram and Whatsapp, to the official Jupyter Lab Gitter, and on Twitter. According to Twitter’s current statistics, the tweet was retweeted 22 times, people saw the invitation tweet 4,049 times, and 326 people interacted with the tweet, despite only 12 filling the initial form – the remaining two participants came from Phase II invitations. Nonetheless, the selection is also biased towards our reach in social media.

**Construct.** Due to the COVID-19 pandemic, we had to design a remote experiment instead of a lab experiment to evaluate Julynter. In this experiment, we distributed Julynter to each participant use at their own pace with their own notebooks. They may have had very distinct usages and different goals that may not justify the usage of Julynter. In fact, during the experiment, some participants used Julynter in scratchpad notebooks that usually do not have a high requirement of quality. Despite this threat, these participants had a positive feeling about Julynter overall.

**External.** We had a small number of participants. Even though we listened to their feedback to improve the tool, the number is not significant to draw conclusions on which are the best recommendations and how users would use the tool in the wild.

## 6.5 Discussion

This chapter introduced a set of best practices for developing notebooks, two noWorkflow integrations for Jupyter, and Julynter as a linting tool to support the best practices. The noWorkflow integrations extend the provenance collection from simple scripts to interactive notebooks, providing all the provenance's underlying benefits. While the noWorkflow extension provides features for collecting individual cell's provenance and analyzing the provenance, the noWorkflow kernel collects provenance from all cell executions and enables notebook cleaning as a provenance application.

While the noWorkflow extensions have benefits, they have the drawback of an extra provenance collection and storage overhead that may not appeal to users. Hence, we also introduced Julynter as the first linting tool that considers the notebook structure. Julynter uses simple provenance data collected by the IPython kernel and attempts to improve the quality and reproducibility of notebooks by recommending actions related to the best practices. We evaluated Julynter with users in a remote experiment and received positive feedback about it.

# Chapter 7

## Conclusion

### 7.1 Contributions

This thesis hypothesizes that scripts and interactive notebooks can also be supported by an infrastructure for collecting, managing, and analyzing provenance from experiments. To support this hypothesis, we improved the noWorkflow’s provenance collection by replacing the passive monitoring strategy with an overriding strategy that allowed us to collect fine-grained provenance from scripts with mutable data structures. We also created an extension and a kernel for provenance collection in interactive notebooks. Moreover, we proposed a command for exporting the provenance as Versioned-PROV and a version model with semantic versions. While we focus on scripts and interactive notebooks in this thesis, the version model could also describe the versioning of workflow experiments. Similarly, the Versioned-PROV definition is generic enough to describe mutable data structures in any PROV document.

We also improved the provenance analysis by introducing a notebook extension, a web visualization tool, and extending the querying and visualization features of noWorkflow by proposing pattern matching, object queries, dataflow graphs, and activation graph summarizing and comparison. For notebooks, we evaluated the usability of Julynter with users in a remote experiment and received positive feedback about it.

The change in the collection strategy may also have improved the efficiency since the overriding strategy only triggers the provenance collection in points of interest. In contrast, the passive monitoring strategy triggers the provenance collection during the execution of function calls in imported modules, and noWorkflow 0 adds an overhead for ignoring the trigger. We did not properly evaluate this effect since both strategies collect provenance at different depth and granularity. Nonetheless, we also improved the efficiency of module collection by collecting modules on demand, and we improved the management efficiency by reducing the storage over-

head. In notebooks, we guaranteed the efficiency of provenance usage by proposing Julynter, a linting tool that considers the structure of notebooks and uses simple provenance information that is readily available in the IPython kernel.

The infrastructure we designed for noWorkflow has also been successfully used in other research projects for algorithmic debugging (LINHARES et al., 2019), and incremental execution (HU et al., 2020). We are also aware that it is also being used in other unpublished research projects to support statistical debugging, test case generation, speed-up of experiment, and collaborative experiments.

During the development of the thesis, we noticed the necessity of better understanding the problem. Hence, we dedicate a good amount of time studying approaches that collect provenance from scripts and understanding how scientists use scripts and notebooks. Based on 27 approaches that collect provenance from scripts, we proposed a taxonomy and categorized them accordingly. We found that very few approaches collect fine-grained provenance that includes variable dependencies, and none of them describe an efficient collection of mutable data structures, as we do in noWorkflow 2. This is important in scripts because many scripts write on mutable data structures. In fact, we found that 55.5% (492,659) of the Python notebooks with valid syntax had at least one assignment to a mutable object attribute or mutable collection item. Moreover, the only approaches that provide mechanisms for analyzing the evolution are the ones that rely on Git generic analyses. They do not offer provenance-tailored evolution analyses.

When we studied how scientists use scripts, we found Python as the favorite tool among most participants who answered a questionnaire. We also noticed that scripting languages have almost no role in provenance and reproducibility when compared to workflow management systems. Studying actual scientific scripts from DataOne, we also found no evidence of provenance usage. We extracted a common structure that scientific scripts follow, and we uncovered the usage rate of each Python construct in scripts. These rates are important to prioritize the features that tool developers should implement for a proper provenance collection.

Since many scientists use interactive notebooks, we analyzed over one million notebooks from GitHub, observing good and bad practices regarding the quality and reproducibility of notebooks. As good practices, we found the usage of literate programming aspects of notebooks (e.g., Markdown cells and visualizations), the application of abstractions on notebooks with more complex control flows, and the usage of descriptive filenames. As bad practices, we found that most notebooks do not test their code. A large number of notebooks have characteristics that hinder the reasoning and reproducibility, such as out-of-order cells, non-executed code

cells, and the possibility of hidden states.

In this study, we also ran a big reproducibility study. We explored distinct execution orders (top-down, execution counter order) and distinct environments (shared environments with installations, isolated environments, and isolated bloated environments with pre-installed dependencies) to run the notebooks. We achieved a reproducibility rate that ranged from 4.90% to 15.04%, indicating that notebooks are not as reproducible as some work usually claim.

## 7.2 Future Work

There are many opportunities for future work to build on top of the contributions of this thesis. With the script analysis' insights, we foresee developing tools and processes targeted at scientists to support the experiments' development. For instance, packages that bundle scientific tools could benefit from knowing which modules scientists use the most and use this information to include these modules in the packages. Additionally, a plugin for an IDE or text editor could provide code snippets based on the conceptual regions we identified (i.e., header, top, definitions, bottom) and processing strategies (i.e., process data during input, during output, in the middle, or interweaving). Finally, this plugin could also provide a "jump" feature to move the cursor directly to each region's beginning.

The analyses we proposed for the script study could also be used for replication with a dataset including scientific scripts from other repositories such as GitHub and Zenodo. Additionally, a reproducibility study could be performed for scripts to compare their reproducibility with the notebooks' reproducibility.

In the notebook reproducibility study, we executed notebooks following distinct execution orders in distinct environments. However, our orders and environments represent only a small subset of the possible configurations. Hence, we foresee reproducibility studies that propose distinct configurations – for instance, executing cells with imports before other cells in an environment created by guessing the repository dependencies. Other reproducibility questions are also worth investigating. We intend to investigate strategies to assess the different types of projects (e.g., student notebooks, tutorial notebooks, research notebooks, scratchpads, dashboards, among others) to compare their metrics.

The infrastructure we built for noWorkflow also has many possibilities for improvements. First, we designed it for collecting provenance from single-threaded scripts. Some assumptions in the AST transformations related to the fine-grained collection do not hold for multi-threaded applications and asynchronous applications. Hence, there is an opportunity of improving it.

Second, the provenance collection is quite limited in terms of black-box function calls. If a function does not have a pure Python definition, `noWorkflow` indicates that its return value was derived from all of its arguments. However, it is not always the case, and users should be able to specify rules for defining dependencies in black-box functions. Similarly, `noWorkflow` considers gray-box functions (i.e., functions that the definition is available outside the user-defined collection depth) as black-box and creates these simplified dependencies as well. There is an opportunity to improve the provenance collection of grey-box functions to perform the full provenance collection but store only the actual dependencies between the return value and the arguments.

Third, the collection of files consider only opening and closing operations, and the provenance encompasses the entire file contents. However, files could be treated as mutable data structures. In this case, there is an opportunity to improve file access collection to answer provenance queries related to which parts of the file contribute to generating the experiment results.

Fourth, the collected dependencies do not distinguish between flows (e.g., argument passing), control dependencies (e.g., if statements), derivations (e.g., binary operations), value assignments (e.g., assignment statements), and same assignments (e.g., singleton assignments) (BOWERS; MCPHILLIPS, T.; LUDÄSCHER, 2018). It could be valuable to distinguish among these types of dependencies for derivation queries. We can currently distinguish some of these types by identifying the dependency name (i.e., argument dependencies have the name `argument`), but it is not consistent for all names.

Finally, there are opportunities to improve the version model to support collaborations, improve queries to support graph provenance queries, and improve storage and monitoring to support real-time provenance analysis.

In addition to the future work in the `noWorkflow` infrastructure itself, the collected provenance also enables other future work related to provenance applications. As stated before, it has been applied for algorithmic debugging, and it is being applied for statistical debugging. However, we also foresee the usage of provenance for an omniscient debugger that allows navigating back in the execution trace.

We foresee the usage of the fine-grained provenance of `noWorkflow` to support code coverage tools. Currently, existing tools use the passive monitoring strategy to collect the activated code in the granularity of lines. `noWorkflow` supports a finer granularity: statements and expressions.

We intend to improve the notebook cleaning interface to allow users to specify the relevant parts of the notebook directly in the Jupyter interface, specifying not only evaluations and code cells but also Markdown cells. We also intend to collect the Markdown provenance in this process. Additionally, we foresee deepening the cleaning to consider only parts of the cells instead of expanding the relevant selection to the whole cell that contains a selected evaluation. Such an algorithm would also allow the creation of dynamic program slices of scripts. With a similar algorithm to the one we use for cleaning, the provenance can also be used to refactor scripts by extracting cells and functions from a code slice.

Currently, the cell dependency collected by Julynter uses only variable name definitions for each cell. This usage is unreliable for dependency detection. We intend to integrate noWorkflow and Julynter to use noWorkflow's dependency collection when its kernel is being used. This integration will improve the cell dependency detection and allow the inclusion of linting suggestions for testing the code and abstracting it into functions, classes, and modules.

Finally, we intend to evaluate the cleaning algorithm using the dataset of the reproducibility study.

## 7.3 Publications and Awards

We published the following papers in the context of this thesis:

- PIMENTEL, J.F.; FREIRE, J.; MURTA, L.; BRAGANHOLO, V. Collecting and analyzing provenance on interactive notebooks: when IPython meets noWorkflow. In: TaPP, 2015.
- PIMENTEL, J.F.; FREIRE, J.; BRAGANHOLO, V.; MURTA, L. Tracking and analyzing the evolution of provenance from scripts. In: IPAW, 2016.
- PIMENTEL, J.F.; FREIRE, J.; MURTA, L.; BRAGANHOLO, V. Fine-grained provenance collection over scripts through program slicing. In: IPAW, 2016.
- PIMENTEL, J.F.; DEY, S.; MCPHILLIPS, T.; BELHAJJAME, K.; KOOP, D.; MURTA L.; BRAGANHOLO, V. Yin & Yang: demonstrating complementary provenance from noWorkflow & YesWorkflow. In: IPAW, 2016.
- PIMENTEL, J.F.; MURTA, L.; BRAGANHOLO, V.; FREIRE, J. noWorkflow: a tool for collecting, analyzing, and managing provenance from python scripts. Proceedings of the VLDB Endowment, 2017.

- PIMENTEL, J.F.; MISSIER, P.; MURTA, L.; BRAGANHOLO, V. Versioned-PROV: A PROV extension to support mutable data entities. In: IPAW, 2018.
- PIMENTEL, J.F.; MURTA, L.; BRAGANHOLO, V.; FREIRE, J. A large-scale study about quality and reproducibility of jupyter notebooks. In: MSR, 2019.
- PIMENTEL, J.F.; FREIRE, J.; MURTA, L.; BRAGANHOLO, V. A survey on collecting, managing, and analyzing provenance from scripts. *ACM Computing Surveys*, 2019.
- PIMENTEL, J.F.; MURTA, L.; BRAGANHOLO, V.; FREIRE, J. Understanding and Improving the Quality and Reproducibility of Jupyter Notebooks. *Empirical Software Engineering*. In press, 2021.

The MSR paper (PIMENTEL et al., 2019b) received two awards:

- ACM SIGSOFT Distinguished Paper Award
- FOSS Impact Paper Award

In addition to these papers, we also published other papers in collaborations that are out of the scope of this thesis during this doctoral course:

- SANTOS, H.; PIMENTEL, J.F.; DA SILVA, V.T.; MURTA, L. Software rejuvenation via a multi-agent approach. *Journal of Systems and Software*, 2015.
- COSTA, C.; FIGUEIREDO, J.J.; PIMENTEL, J.F.; SARMA, A.; MURTA, L. Recommending Participants for Collaborative Merge Sessions. *IEEE Transactions on Software Engineering*, 2019.
- LINHARES, H.; PIMENTEL, J.F.; KOHWALTER, T.; MURTA, L. Provenance-enhanced Algorithmic Debugging. In *SBES*, 2020.
- MOURÃO, E.; PIMENTEL, J.F.; MURTA, L.; KALINOWSKI, M.; MENDES, E.; WOHLIN, C. On the performance of hybrid search strategies for systematic literature reviews in software engineering. *Information and Software Technology*, 2020.
- MENEZES, J.W.; TRINDADE, B.; PIMENTEL, J.F.; MOURA, T.; PLASTINO, A.; MURTA, L.; COSTA, C. What causes merge conflicts? In: *SBES*, 2020.

The SBES 2020 paper (MENEZES et al., 2020) received the Best Paper Award.

# References

- ACUÑA, Ruben. **Understanding Legacy Workflows through Runtime Trace Analysis**. 2015. MA thesis – Arizona State University.
- ACUÑA, Ruben; CHOMILIER, Jacques; LACROIX, Zoé. Managing and Documenting Legacy Scientific Workflows. **Journal of Integrative Bioinformatics**, v. 12, n. 3, p. 277–277, 2015.
- ACUÑA, Ruben; LACROIX, Zoé. Extracting Semantics from Legacy Scientific Workflows. In: ICSC. Laguna Hills, USA: IEEE, 2016. p. 9–16.
- ACUÑA, Ruben; LACROIX, Zoé; BAZZI, Rida A. Instrumentation and Trace Analysis for Ad-Hoc Python Workflows in Cloud Environments. In: CLOUD. New York, USA: IEEE, 2015. p. 114–121.
- ADIDA, Ben; BIRBECK, Mark; MCCARRON, Shane; PEMBERTON, Steven. RDFa in XHTML: Syntax and processing. **W3C Proposed Recommendation**, v. 7, p. 1–89, 2008.
- AGRAWAL, Rakesh; SRIKANT, Ramakrishnan. Fast Algorithms for Mining Association Rules. In: VLDB. Santiago de Chile, Chile: Morgan Kaufmann, 1994. p. 487–499.
- ALTINTAS, Ilkay; BARNEY, Oscar; JAEGER-FRANK, Efrat. Provenance collection support in the kepler scientific workflow system. In: IPAW. Chicago, USA: Springer, 2006. p. 118–132.
- ANACONDA. **Anaconda Software Distribution**. Accessed: 2019-10-01. 2018. Available from: <<https://www.anaconda.com>>.
- ANAND, Manish Kumar; BOWERS, Shawn; LUDÄSCHER, Bertram. Provenance browser: Displaying and querying scientific workflow provenance graphs. In: ICDE. Long Beach, USA: IEEE, 2010. p. 1201–1204.
- ANGELINO, Elaine; BRAUN, Uri; HOLLAND, David A; MARGO, Daniel W. Provenance Integration Requires Reconciliation. In: TAPP. Heraklion, Crete, Greece: USENIX, 2011. p. 1–6.
- ANGELINO, Elaine; YAMINS, Daniel; SELTZER, Margo. StarFlow: A script-centric data analysis environment. In: IPAW. Troy, USA: Springer, 2010. p. 236–250.

- ARNAOUDOVA, Venera; DI PENTA, Massimiliano; ANTONIOL, Giuliano. Linguistic antipatterns: What they are and how developers perceive them. **Empirical Software Engineering**, Springer, v. 21, n. 1, p. 104–158, 2016.
- AUTHORITY, Python Code Quality. **Astroid: A common base representation of python source code for pylint and other projects**. Accessed: 01.24.2020. Dec. 2017. Available from: <<https://github.com/PyCQA/astroid>>.
- BAGGERLY, Keith A; COOMBES, Kevin R. Deriving chemosensitivity from cell lines: Forensic bioinformatics and reproducible research in high-throughput biology. **The Annals of Applied Statistics**, JSTOR, v. 3, n. 4, p. 1309–1334, 2009.
- BANGOR, Aaron; KORTUM, Philip T; MILLER, James T. An empirical evaluation of the system usability scale. **International Journal of Human–Computer Interaction**, Taylor & Francis, v. 24, n. 6, p. 574–594, 2008.
- BAO, Zhuowei; COHEN-BOULAKIA, Sarah; DAVIDSON, Susan B; GIRARD, Pierrick. PDiffView: viewing the difference in provenance of workflow results. **Proceedings of the VLDB Endowment**, v. 2, n. 2, p. 1638–1641, 2009.
- BARKER, Adam; VAN HEMERT, Jano. Scientific workflow: a survey and research directions. In: PPAM. Gdansk, Poland: Springer, 2007. p. 746–753.
- BECKER, Gabriel; MOORE, Sara E; LAWRENCE, Michael. trackr: A Framework for Enhancing Discoverability and Reproducibility of Data Visualizations and Other Artifacts in R. **Journal of Computational and Graphical Statistics**, Taylor & Francis, v. 28, n. 3, p. 644–658, 2019.
- BECKER, Richard A; CHAMBERS, John M. Auditing of data analyses. **SIAM Journal on Scientific and Statistical Computing**, SIAM, v. 9, n. 4, p. 747–760, 1988.
- BENEDEK, Joey; MINER, Trish. Measuring Desirability: New methods for evaluating desirability in a usability lab setting. **Proceedings of Usability Professionals Association**, v. 2003, n. 8-12, p. 57, 2002.
- BOCHNER, Carsten; GUDE, Roland; SCHREIBER, Andreas. A python library for provenance recording and querying. In: IPAW. Salt-Lake City, USA: Springer, 2008. p. 229–240.
- BOWERS, Shawn; MCPHILLIPS, Timothy M; LUDÄSCHER, Bertram. Provenance in collection-oriented scientific workflows. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 20, n. 5, p. 519–529, 2008.

- BOWERS, Shawn; MCPHILLIPS, Timothy; LUDÄSCHER, Bertram. Validation and Inference of Schema-Level Workflow Data-Dependency Annotations. In: IPAW. London, United Kingdom: Springer, 2018. p. 128–141.
- BRACHMANN, Michael; SPOTH, William. Your notebook is not crumbly enough, REPLace it. In: CIDR. Amsterdam, The Netherlands: www.cidrdb.org, 2020.
- BRACHMANN, Mike; BAUTISTA, Carlos; CASTELO, Sonia; FENG, Su; FREIRE, Juliana; GLAVIC, Boris; KENNEDY, Oliver; MÜELLER, Heiko; RAMPIN, Rémi; SPOTH, William; YANG, Ying. Data debugging and exploration with vizier. In: SIGMOD. Amsterdam, The Netherlands: ACM, 2019. p. 1877–1880.
- BRAUN, Uri; GARFINKEL, Simson; HOLLAND, David A; MUNISWAMY-REDDY, Kiran-Kumar; SELTZER, Margo I. Issues in automatic provenance collection. In: IPAW. Chicago, USA: Springer, 2006. p. 171–183.
- BROOKE, John. SUS: a “quick and dirty” usability. **Usability Evaluation in Industry**, CRC press, p. 189, 1996.
- CALLAHAN, Steven P; FREIRE, Juliana; SANTOS, Emanuele; SCHEIDEGGER, Carlos Eduardo; SILVA, Claudio T; VO, Huy T. Managing the Evolution of Dataflows with VisTrails. In: ICDE. Atlanta, USA: IEEE, 2006. p. 71–71.
- CANNON, Brett; SMITH, Nathaniel; STUFFT, Donald. **PEP 518: Specifying Minimum Build System Requirements for Python Projects**. Accessed: 2020-09-22. 2016. Available from: <<https://www.python.org/dev/peps/pep-0518/>>.
- CARVALHO, L; BELHAJJAME, Khalid; MEDEIROS, C. A PROV-compliant approach to script-to-workflow process. **The Semantic Web Journal**, 2018.
- CHACON, Scott; STRAUB, Ben. **Pro Git**. 2. ed. New York, USA: Apress, 2014.
- CHAMBERS, John M. **Programming with Data: A Guide to the S Language**. New York, USA: Springer Science & Business Media, 1998.
- CHAPMAN, Adriane; JAGADISH, HV. Understanding provenance black boxes. **Distributed and Parallel Databases**, Springer, v. 27, n. 2, p. 139–167, 2010.
- CHAPMAN, Adriane; MISSIER, Paolo; SIMONELLI, Giulia; TORLONE, Riccardo. Capturing and Querying Fine-grained Provenance of Preprocessing Pipelines in Data Science. **Proceedings of the VLDB Endowment**, 2021.
- CHAVAN, Amit; HUANG, Silu; DESHPANDE, Amol; ELMORE, Aaron; MADDEN, Samuel; PARAMESWARAN, Aditya. Towards a unified query language for provenance and versioning. In: TAPP. Edinburgh, Scotland: USENIX, 2015. p. 1–6.

- CHEBOTKO, Artem; ABRAHAM, John; BRAZIER, Pearl; PIAZZA, Anthony; KASHLEV, Andrey; LU, Shiyong. Storing, indexing and querying large provenance data sets as RDF graphs in apache HBase. In: SERVICES. Santa Clara, USA: IEEE, 2013. p. 1–8.
- CHEBOTKO, Artem; LU, Shiyong; FEI, Xubo; FOTOUHI, Farshad. RDFProv: A relational RDF store for querying and managing scientific workflow provenance. **Data & Knowledge Engineering**, Elsevier, v. 69, n. 8, p. 836–865, 2010.
- CHENEY, James; AHMED, Amal; ACAR, Umut A. Provenance as dependency analysis. **Mathematical Structures in Computer Science**, Cambridge University Press, v. 21, n. 6, p. 1301–1337, 2011.
- CHENEY, James; CHITICARIU, Laura; TAN, Wang-Chiew. Provenance in Databases: Why, How, and Where. **Foundations and Trends in Databases**, v. 1, n. 4, p. 379–474, 2007.
- CHIRIGATI, Fernando; RAMPIN, Rémi; SHASHA, Dennis; FREIRE, Juliana. Rezip: Computational reproducibility with ease. In: SIGMOD. San Francisco, USA: ACM, 2016. p. 2085–2088.
- CHIRIGATI, Fernando; SHASHA, Dennis; FREIRE, Juliana. Rezip: Using provenance to support computational reproducibility. In: TAPP. Lombard, USA: USENIX, 2013. p. 977–980.
- CHITTIMALLI, Pavan Kumar; NAIK, Ravindra. Variable provenance in software systems. In: RSSE. Hyderabad, India: ACM, 2014. p. 9–13.
- CLAERBOUT, Jon; KARRENBACH, Martin. Electronic documents give reproducible research a new meaning. In: SEG. New Orleans, USA: SEG, 1992. p. 601–604.
- CLIFF, Norman. Answering ordinal questions with ordinal data using ordinal statistics. **Multivariate Behavioral Research**, Taylor & Francis, v. 31, n. 3, p. 331–350, 1996.
- CLIFFORD, Ben; FOSTER, Ian; VOECKLER, Jens-S; WILDE, Michael; ZHAO, Yong. Tracking provenance in a virtual data grid. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 20, n. 5, p. 565–575, 2008.
- COLLBERG, Christian; PROEBSTING, Todd; MORAILA, Gina; SHANKARAN, Akash; SHI, Zuoming; WARREN, Alex M. **Measuring reproducibility in computer systems research**. Department of Computer Science, University of Arizona, 2014. p. 1–37.
- CONRADI, Reidar; WESTFECHTEL, Bernhard. Version models for software configuration management. **ACM Computing Surveys**, ACM, v. 30, n. 2, p. 232–282, 1998.
- COOK, John. **Code to slice open a Menger sponge**. Accessed: 2021-05-03. Available from: <<https://www.johndcook.com/blog/2011/08/30/slice-a-menger-sponge/>>.

- COSTA, Flavio; SILVA, Vítor; DE OLIVEIRA, Daniel; OCAÑA, Kary; OGASAWARA, Eduardo; DIAS, Jonas; MATTOSO, Marta. Capturing and querying workflow runtime provenance with PROV: a practical approach. In: EDBT/ICDT. Genoa, Italy: ACM, 2013. p. 282–289.
- CRUZ, Sergio Manuel Serra da; NASCIMENTO, José Antonio Pires do. SisGExp: rethinking long-tail agronomic experiments. In: IPAW. McLean, USA: Springer, 2016. p. 214–217.
- DA CRUZ, Sérgio Manuel Serra; NASCIMENTO, José Antonio Pires do. Towards integration of data-driven agronomic experiments with data provenance. **Computers and Electronics in Agriculture**, Elsevier, v. 161, p. 14–28, 2019.
- DAVIDSON, Susan B; FREIRE, Juliana. Provenance and scientific workflows: challenges and opportunities. In: SIGMOD. Vancouver, BC, Canada: ACM, 2008. p. 1345–1350.
- DAVISON, Andrew. Automated capture of experiment context for easier reproducibility in computational research. **Computing in Science & Engineering**, AIP Publishing, v. 14, n. 4, p. 48–56, 2012.
- DEMSKY, Brian. Garm: cross application data provenance and policy enforcement. In: HOTSEC. Montreal, Canada: USENIX, 2009. v. 9, p. 10–10.
- DEY, Saumen; BELHAJJAME, Khalid; KOOP, David; RAUL, Meghan; LUDÄSCHER, Bertram. Linking prospective and retrospective provenance in scripts. In: TAPP. Edinburgh, Scotland: USENIX, 2015. p. 1–7.
- DIAS, Jonas; OGASAWARA, Eduardo; OLIVEIRA, Daniel de; PORTO, Fabio; COUTINHO, Alvaro LGA; MATTOSO, Marta. Supporting dynamic parameter sweep in adaptive and user-steered workflow. In: WORKS. Seattle, USA: ACM, 2011. p. 31–36.
- DIAS, Luiz Gustavo; MATTOSO, Marta; LOPES, Bruno; OLIVEIRA, Daniel de. Experiencing DfAnalyzer for Runtime Analysis of Phylogenomic Dataflows. In: BSB. São Paulo, Brazil: Springer, 2020. p. 105–116.
- DIETRICH, Christian; LOHMANN, Daniel. The dataref versuchung: Saving time through better internal repeatability. **SIGOPS Operating Systems Review**, ACM, v. 49, n. 1, p. 51–60, 2015.
- DONOHO, David L; MALEKI, Arian; RAHMAN, Inam Ur; SHAHRAM, Morteza; STODDEN, Victoria. Reproducible research in computational harmonic analysis. **Computing in Science & Engineering**, AIP Publishing, v. 11, n. 1, p. 8–18, 2009.
- DRUMMOND, Chris. Replicability is not reproducibility: nor is it good science. In: ICML. Montreal, CA: International Machine Learning Society, 2009. p. 1–4.

- DUBOIS, Paul F. Guest Editor's Introduction: Python–Batteries Included. **Computing in Science & Engineering**, IEEE Computer Society, v. 9, n. 3, p. 7–9, 2007.
- DUBOIS, Paul F. Ten good practices in scientific programming. **Computing in Science & Engineering**, AIP Publishing, v. 1, n. 1, p. 7–11, 1999.
- EICHINSKI, Philip; ROE, Paul. Datatrack: An R package for managing data in a multi-stage experimental workflow. In: ESON. Baltimore, USA: IEEE, 2016. p. 1–8.
- ESTUBLIER, Jacky. Software Configuration Management: A Roadmap. In: ICSE. New York, USA: ACM, 2000. p. 279–289. ISBN 1-58113-253-0.
- FELIZARDO, Katia Romero; MENDES, Emilia; KALINOWSKI, Marcos; SOUZA, Érica Ferreira; VIJAYKUMAR, Nandamudi L. Using forward snowballing to update systematic reviews in software engineering. In: ESEM. Ciudad Real, Spain: ACM, 2016. p. 1–6.
- FILGUIERA, Rosa; KLAMPANOS, Iraklis; KRAUSE, Amrey; DAVID, Mario; MORENO, Alexander; ATKINSON, Malcolm. Dispel4Py: A Python Framework for Data-intensive Scientific Computing. In: DATA CLOUD@SC. Salt Lake, USA: IEEE Computer Society, 2014. p. 9–16.
- FILGUIERA, Rosa; KRAUSE, Amrey; ATKINSON, Malcolm; KLAMPANOS, Iraklis; MORENO, Alexander. dispel4py: A python framework for data-intensive scientific computing. **The International Journal of High Performance Computing Applications**, SAGE Publications Sage UK: London, England, v. 31, n. 4, p. 316–334, 2017.
- FLOYD, Robert W. Algorithm 97: shortest path. **Commun. ACM.**, ACM, v. 5, n. 6, p. 345, 1962.
- FREIRE, Juliana; KOOP, David; SANTOS, Emanuele; SILVA, Cláudio T. Provenance for computational tasks: A survey. **Computing in Science & Engineering**, AIP Publishing, v. 10, n. 3, p. 11–21, 2008.
- FREIRE, Juliana; SILVA, Cláudio T; CALLAHAN, Steven P; SANTOS, Emanuele; SCHEIDEGGER, Carlos E; VO, Huy T. Managing rapidly-evolving scientific workflows. In: IPAW. Chicago, USA: Springer, 2006. p. 10–18.
- FREW, James. Earth System Science Server (ES3): Local Infrastructure for Earth Science Product Management. In: ESTC. Palo Alto, CA: NASA, 2004. p. 1–5.
- FREW, James; BOSE, Rajendra. Earth system science workbench: A data management infrastructure for earth science products. In: SSDBM. Fairfax, VA, U.S.A: IEEE, 2001. p. 180–189.

- FREW, James; JANÉE, Greg; SLAUGHTER, Peter. Automatic Provenance Collection and Publishing in a Science Data Production Environment – Early Results. In: IPAW. Troy, USA: Springer, 2010. p. 27–33.
- FREW, James; JANÉE, Greg; SLAUGHTER, Peter. Provenance-enabled automatic data publishing. In: SSDBM. Portland, USA: Springer, 2011. p. 244–252.
- FREW, James; METZGER, Dominic; SLAUGHTER, Peter. Automatic capture and reconstruction of computational provenance. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 20, n. 5, p. 485–496, 2008.
- FREW, James; SLAUGHTER, Peter. Es3: A demonstration of transparent provenance for scientific computation. In: IPAW. Salt Lake City, USA: Springer, 2008. p. 200–207.
- GAILLY, Jean-loup; ADLER, Mark. **zlib**. Accessed: 2021-01-07. 2017. Available from: <<https://github.com/madler/zlib>>.
- GARIJO, Daniel; GIL, Yolanda. Augmenting PROV with Plans in P-PLAN: Scientific Processes as Linked Data. In: LISC. Boston, USA: CEUR-WS.org, 2012.
- GAROUSHI, Vahid; KÜÇÜK, Barış. Smells in software test code: A survey of knowledge in industry and academia. **Journal of Systems and Software**, Elsevier, v. 138, p. 52–81, 2018.
- GAVISH, Matan; DONOHO, David. A universal identifier for computational results. **Procedia Computer Science**, Elsevier, v. 4, p. 637–647, 2011.
- GELFOND, Jonathan; GOROS, Martin; HERNANDEZ, Brian; BOKOV, Alex. A system for an accountable data analysis process in R. **The R Journal**, NIH Public Access, v. 10, n. 1, p. 6, 2018.
- GHARIBI, Gharib; WALUNJ, Vijay; ALANAZI, Rakan; RELLA, Sirisha; LEE, Yugyung. Automated management of deep learning experiments. In: DEEM@SIGMOD. Amsterdam, The Netherlands: ACM, 2019. p. 1–4.
- GILL, Richard D. Event based simulation of an EPR-B experiment by local hidden variables: epr-simple and epr-clocked. **arXiv preprint arXiv:1507.00106**, 2015.
- GLAVIC, Boris; DITTRICH, Klaus R. Data Provenance: A Categorization of Existing Approaches. In: BTW. Aachen, Germany: GI, 2007. p. 227–241.
- GREFF, Klaus; KLEIN, Aaron; CHOVANEC, Martin; HUTTER, Frank; SCHMIDHUBER, Jürgen. The sacred infrastructure for computational research. In: SCIPY. Austin, USA: SciPy Conference, 2017. v. 28, p. 49–56.

- GREFF, Klaus; SCHMIDHUBER, Jürgen. Introducing Sacred: A Tool to Facilitate Reproducible Research. In: AUTOML. Lille, France: International Machine Learning Society, 2015. p. 1–6.
- GROTH, Paul; MILES, Simon; MOREAU, Luc. PReServ: Provenance recording for services. In: UK e-Science All Hands Meeting. Nottingham, UK: EPSRC, 2005. v. 2005, p. 1–8.
- GRUS, Joel. **I don't like notebooks**. JupyterCon Presentation. Accessed: 2021-02-19. 2018. Available from: <<https://conferences.oreilly.com/jupyter/jup-public/schedule/detail/68282>>.
- GUO, Philip J; ENGLER, Dawson. Using automatic persistent memoization to facilitate data analysis scripting. In: ISSTA. Toronto, ON, Canada: ACM, 2011. p. 287–297.
- GUO, Philip J; ENGLER, Dawson R. CDE: Using System Call Interposition to Automatically Create Portable Software Packages. In: ATC. Portland, USA: USENIX Association, 2011. p. 1–6.
- GUO, Philip J; ENGLER, Dawson R. Towards Practical Incremental Recomputation for Scientists: An Implementation for the Python Language. In: IPAW. Troy, USA: Springer, 2010. p. 1–10.
- GUO, Philip J; SELTZER, Margo. BURRITO: Wrapping Your Lab Notebook in Computational Infrastructure. In: TAPP. Boston, USA: USENIX, 2012. v. 12, p. 1–7.
- GUO, Philip Jia. **Software tools to facilitate research programming**. 2012. PhD thesis – Stanford University, Stanford University.
- HAN, Jiawei; PEI, Jian; KAMBER, Micheline. **Data mining: concepts and techniques**. Waltham, USA: Morgan Kaufmann, 2011.
- HAN, Peiyi; WANG, Chaozheng; LIU, Chuanyi; DUAN, Shaoming; PAN, Hezhong; LUO, Pengshuai. SecureMLDebugger: A Privacy-Preserving Machine Learning Debugging Tool. In: DSC. Hong Kong, China: IEEE, 2020. p. 127–134.
- HANNAY, Jo Erskine; MACLEOD, Carolyn; SINGER, Janice; LANGTANGEN, Hans Petter; PFAHL, Dietmar; WILSON, Greg. How do scientists develop and use scientific software? In: SE-CSE. Vancouver, BC, Canada: IEEE Computer Society, 2009. p. 1–8.
- HANSON, Brooks; SUGDEN, Andrew; ALBERTS, Bruce. Making data maximally available. **Science**, American Association for the Advancement of Science, New York, N.Y., v. 331, n. 6018, p. 649–649, 2011.
- HEAD, Andrew. **Interactive Program Distillation**. 2020. PhD thesis – UC Berkeley, UC Berkeley.

- HEAD, Andrew; HOHMAN, Fred; BARIK, Titus; DRUCKER, Steven M; DELINE, Robert. Managing messes in computational notebooks. In: CHI. Glasgow, Scotland, UK: ACM, 2019. p. 1–12.
- HEDGES, Larry V; OLKIN, Ingram. **Statistical methods for meta-analysis**. Waktham, USA: Academic press, 2014.
- HERSCHEL, Melanie; DIESTELKÄMPER, Ralf; LAHMAR, Housseem Ben. A survey on provenance: What for? What form? What from? **VLDB Journal**, v. 26, n. 6, p. 881–906, 2017.
- HIRSCHBERG, Daniel S. Algorithms for the longest common subsequence problem. **Journal of the ACM**, ACM, v. 24, n. 4, p. 664–675, 1977.
- HOBAN, Sean; BERTORELLE, Giorgio; GAGGIOTTI, Oscar E. Computer simulations: tools for population and evolutionary genetics. **Nature Reviews Genetics**, Nature Publishing Group, v. 13, n. 2, p. 110–122, 2012.
- HOEKSTRA, Rinke; GROTH, Paul. PROV-O-Viz-understanding the role of activities in provenance. In: IPAW. Cologne, Germany: Springer, 2014. p. 215–220.
- HORWITZ, Susan; REPS, Thomas. The use of program dependence graphs in software engineering. In: ICSE. Melbourne, Australia: ACM, 1992. p. 392–411.
- HU, Jingmei; JOUNG, Jiwon; JACOBS, Maia; GAJOS, Krzysztof Z; SELTZER, Margo I. Improving data scientist efficiency with provenance. In: ICSE. Seoul, South Korea: ACM, 2020. p. 1086–1097.
- HUQ, Mohammad Rezwanul. **An inference-based framework for managing data provenance**. 2013. PhD thesis – University of Twente.
- HUQ, Mohammad Rezwanul; APERS, Peter MG; WOMBACHER, Andreas. An inference-based framework to manage data provenance in Geoscience Applications. **IEEE Transactions on Geoscience and Remote Sensing**, IEEE, Washington, USA, v. 51, n. 11, p. 5113–5130, 2013a.
- HUQ, Mohammad Rezwanul; APERS, Peter MG; WOMBACHER, Andreas. ProvenanceCurious: a tool to infer data provenance from scripts. In: EDBT. Genoa, Italy: ACM, 2013b. p. 765–768.
- HÜRSCH, Walter L; LOPES, Cristina Videira. **Separation of Concerns**. Northeastern University, USA, 1995.
- IBÁÑEZ, J. David et al. **pygit2**. Accessed: 2021-01-07. 2018. Available from: <<https://github.com/libgit2/pygit2>>.

- IOANNIDIS, John PA. Why most published research findings are false. **PLOS Medicine**, Public Library of Science, v. 2, n. 8, e124, 2005.
- ISRAEL, Glenn D. **Determining sample size**. University of Florida, USA, 1992.
- IVES, Zack; ZHANG, Yi; HAN, Soonbo; ZHENG, Nan. Dataset Relationship Management. In: CIDR. Asilomar, USA: www.cidrdb.org, 2019.
- IVIE, Peter. **A Workflow Management System to Facilitate Reproducibility of Scientific Computing Applications**. 2018. PhD thesis – University of Notre Dame, University of Notre Dame.
- JACKSON, Keith R. pyGlobus: a Python interface to the Globus Toolkit™. **Concurrency and Computation: Practice and Experience** – 13–15, Wiley Online Library, v. 14, n. 13–15, p. 1075–1083, 2002.
- JALALI, Samireh; WOHLIN, Claes. Systematic literature studies: database searches vs. backward snowballing. In: ESEM. Lund University, Sweden: ACM, 2012. p. 29–38.
- JONES, Matthew B; LUDÄSCHER, Bertram; MCPHILLIPS, Timothy; MISSIER, Paolo; SCHWALM, Christopher; SLAUGHTER, Peter; VIEGLAIS, Dave; WALKER, Lauren; WEI, Yaxing. DataONE: A Data Federation with Provenance Support. In: IPAW. McLean, USA: Springer, 2016. v. 9672, p. 230.
- KALLIAMVAKOU, Eirini; GOUSIOS, Georgios; BLINCOE, Kelly; SINGER, Leif; GERMAN, Daniel M; DAMIAN, Daniela. The promises and perils of mining GitHub. In: MSR. Hyderabad, India: ACM, 2014. p. 92–101.
- KERY, Mary Beth. Tools to support exploratory programming with data. In: VL/HCC. Raleigh, USA: IEEE, 2017. p. 321–322.
- KERY, Mary Beth; HORVATH, Amber; MYERS, Brad. Variolite: Supporting Exploratory Programming by Data Scientists. In: CHI. Denver, USA: ACM, 2017. p. 1–12.
- KERY, Mary Beth; JOHN, Bonnie E; O’FLAHERTY, Patrick; HORVATH, Amber; MYERS, Brad A. Towards effective foraging by data scientists to find past analysis choices. In: CHI. Glasgow, Scotland, UK: ACM, 2019. p. 1–13.
- KERY, Mary Beth; MYERS, Brad A. Interactions for untangling messy history in a computational notebook. In: VL/HCC. Lisbon, Portugal: IEEE, 2018. p. 147–155.
- KERY, Mary Beth; RADENSKY, Marissa; ARYA, Mahima; JOHN, Bonnie E.; MYERS, Brad A. The Story in the Notebook: Exploratory Data Science Using a Literate Programming Tool. In: CHI. Montreal QC, Canada: ACM, 2018. 174:1–174:11. ISBN 978-1-4503-5620-6. DOI: 10.1145/3173574.3173748.

- KIM, Jihie; DEELMAN, Ewa; GIL, Yolanda; MEHTA, Gaurang; RATNAKAR, Varun. Provenance trails in the wings/pegasus system. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 20, n. 5, p. 587–597, 2008.
- KLUYVER, Thomas; RAGAN-KELLEY, Benjamin; PÉREZ, Fernando; GRANGER, Brian E; BUSSONNIER, Matthias; FREDERIC, Jonathan; KELLEY, Kyle; HAMRICK, Jessica B; GROUT, Jason; CORLAY, Sylvain; IVANOV, Paul; AVILA, Damián; ABDALLA, Safia; WILLING, Carol, et al. Jupyter Notebooks - a publishing format for reproducible computational workflows. In: ELPUB. Göttingen, Germany: IOS Press, 2016. p. 87–90.
- KNUTH, Donald E. Literate programming. **Computer**, v. 1, n. 2, p. 97–111, 1984.
- KOENZEN, Andreas; ERNST, Neil; STOREY, Margaret-Anne. Code Duplication and Reuse in Jupyter Notebooks. **arXiv preprint arXiv:2005.13709**, 2020.
- KOHWALTER, Troy; OLIVEIRA, Thiago; FREIRE, Juliana; CLUA, Esteban; MURTA, Leonardo. Prov Viewer: a graph-based visualization tool for interactive exploration of provenance data. In: IPAW. McLean, USA: Springer, 2016. p. 71–82.
- KOOP, David; PATEL, Jay. Dataflow notebooks: encoding and tracking dependencies of cells. In: TAPP. Seattle, Washington: USENIX, 2017. p. 1–7.
- KOOP, David; SANTOS, Emanuele; BAUER, Bela; TROYER, Matthias; FREIRE, Juliana; SILVA, Cláudio T. Bridging workflow and data provenance using strong links. In: SSDBM. Portland, USA: Springer, 2010. v. 28, p. 397–415.
- KOSLOW, Stephen H. Sharing primary data: a threat or asset to discovery? **Nature Reviews Neuroscience**, Nature Publishing Group, v. 3, n. 4, p. 311, 2002.
- KÖSTER, Johannes; RAHMANN, Sven. Snakemake—a scalable bioinformatics workflow engine. **Bioinformatics**, Oxford Univ Press, v. 28, n. 19, p. 2520–2522, 2012.
- LANGTANGEN, Hans Petter. **Python scripting for computational science**. 3rd. Berlin, Heidelberg and New York: Springer, 2006. v. 3.
- LERNER, Barbara; BOOSE, Emery. POSTER: RDataTracker and DDG Explorer. In: IPAW. Cologne, Germany: Springer, 2014a. p. 1–3.
- LERNER, Barbara; BOOSE, Emery. RDataTracker: collecting provenance in an interactive scripting environment. In: TAPP. Cologne, Germany: USENIX, 2014b. p. 1–4.
- LERNER, Barbara; BOOSE, Emery; PEREZ, Luis. Using Introspection to Collect Provenance in R. **Informatics**, Multidisciplinary Digital Publishing Institute, v. 5, n. 1, p. 12, 2018.

- LEWINE, Donald. **POSIX programmers guide**. Sebastopol, USA: "O'Reilly Media, Inc.", 1991.
- LIM, Chunhyeok; LU, Shiyong; CHEBOTKO, Artem; FOTOUHI, Farshad. Prospective and retrospective provenance collection in scientific workflow environments. In: SCC. Miami, USA: IEEE, 2010. p. 449–456.
- LIM, Chunhyeok; LU, Shiyong; CHEBOTKO, Artem; FOTOUHI, Farshad;  
KASHLEV, Andrey. OPQL: querying scientific workflow provenance at the graph level. **Data and Knowledge Engineering**, Elsevier, v. 88, n. 0, p. 37–59, 2013.
- LIN, Cui; LU, Shiyong; FEI, Xubo; CHEBOTKO, Artem; PAI, Darshan; LAI, Zhaoqiang; FOTOUHI, Farshad; HUA, Jing. A reference architecture for scientific workflow management systems and the VIEW SOA solution. **IEEE Transactions on Services Computing**, IEEE, v. 2, n. 1, p. 79–92, 2009.
- LINHARES, Henrique; PIMENTEL, João Felipe; KOHWALTER, Troy;  
MURTA, Leonardo Gresta Paulino. Provenance-enhanced Algorithmic Debugging. In: SBES. Salvador, Brazil: ACM, 2019. p. 203–212.
- LIU, Ji; PACITTI, Esther; VALDURIEZ, Patrick; MATTOSO, Marta. A Survey of Data-Intensive Scientific Workflow Management. **Journal of Grid Computing**, Springer, v. 13, n. 4, p. 457–493, 2015.
- LOUI, Ronald P. In praise of scripting: Real programming pragmatism. **Computer**, IEEE, v. 41, n. 7, p. 22–26, 2008.
- LYNCH, Clifford. Authenticity and integrity in the digital environment: an exploratory analysis of the central role of trust. **CLIR**, June, v. 32, n. 1, p. 1–84, 2000.
- MACKO, Peter; SELTZER, Margo. A General-Purpose Provenance Library. In: TAPP. Boston, USA: USENIX, 2012. p. 1–6.
- MARINHO, Anderson; MATTOSO, Marta; WERNER, Claudia; BRAGANHOLO, Vanessa; MURTA, Leonardo. Challenges in Managing Implicit and Abstract Provenance Data: Experiences with ProvManager. In: TAPP. Heraklion, Crete, Greece: USENIX, 2011. p. 1–6.
- MARTIN, Andrew P; LYLE, John; NAMILUKO, Cornelius. Provenance as a Security Control. In: TAPP. Boston, USA: USENIX, 2012.
- MATTOSO, Marta; DIAS, Jonas; OCAÑA, Kary ACS; OGASAWARA, Eduardo;  
COSTA, Flavio; HORTA, Felipe; SILVA, Vítor; OLIVEIRA, Daniel de. Dynamic steering of HPC scientific workflows: A survey. **Future Generation Computer Systems**, Elsevier, v. 46, p. 100–113, 2015.

- MATTOSO, Marta; WERNER, Claudia; TRAVASSOS, Guilherme Horta; BRAGANHOLO, Vanessa; OGASAWARA, Eduardo; OLIVEIRA, Daniel; CRUZ, Sergio; MARTINHO, Wallace; MURTA, Leonardo. Towards supporting the life cycle of large scale scientific experiments. **International Journal of Business Process Integration and Management**, Inderscience Publishers, v. 5, n. 1, p. 79–92, 2010.
- MCFEE, Brian; RAFFEL, Colin; LIANG, Dawen; ELLIS, Daniel PW; MCVICAR, Matt; BATTENBERG, Eric; NIETO, Oriol. librosa: Audio and music signal analysis in python. In: SCIPY. Austin, USA: SciPy Conference, 2015. v. 8, p. 18–25.
- MCGUGAN, Will. **Beginning game development with Python and Pygame: from novice to professional**. New York, USA: Apress, 2007.
- MCKINNEY, Wes. pandas: a foundational Python library for data analysis and statistics. **Python for High Performance and Scientific Computing**, v. 14, n. 9, 2011.
- MCPHILLIPS, Timothy; BOWERS, Shawn; BELHAJJAME, Khalid; LUDÄSCHER, Bertram. Retrospective provenance without a runtime provenance recorder. In: TAPP. Edinburgh, Scotland: USENIX, 2015a. p. 1–7.
- MCPHILLIPS, Timothy; SONG, Tianhong; KOLISNIK, Tyler; AULENBACH, Steve; BELHAJJAME, Khalid; BOCINSKY, Kyle; CAO, Yang; CHENEY, James; CHIRIGATI, Fernando; DEY, Saumen; FREIRE, Juliana; JONES, Christopher; HANKEN, James; KINTIGH, Keith; KOHLER, Timothy; KOOP, David; MACKLIN, James; MISSIER, Paolo; SCHILDHAUER, Mark; SCHWALM, Christopher; WEI, Yaxing; BIEDA, Mark; LUDÄSCHER, Bertram. YesWorkflow: a user-oriented, language-independent tool for recovering workflow information from scripts. **International Journal of Digital Curation**, v. 10, n. 1, p. 298–313, 2015b.
- MENEZES, José William; TRINDADE, Bruno; PIMENTEL, João Felipe; MOURA, Tayane; PLASTINO, Alexandre; MURTA, Leonardo; COSTA, Catarina. What causes merge conflicts? In: SBES. Natal, Brazil: ACM, 2020. p. 203–212.
- MEYER, Robert. **Correlations and coding in visual cortex**. 2016. MA thesis – Technische Universitaet Berlin, Germany.
- MEYER, Robert; OBERMAYER, Klaus. pypet: A Python Toolkit for Data Management of Parameter Explorations. **Frontiers in Neuroinformatics**, Frontiers Media SA, v. 10, p. 1–16, 2016.
- MEYER, Robert; OBERMAYER, Klaus. pypet: a python toolkit for simulations and numerical experiments. **Neuroscience**, BioMed Central, v. 16, Suppl 1, p184, 2015.

- MICHAELIDES, Danius T; PARKER, Richard; CHARLTON, Chris; BROWNE, William J; MOREAU, Luc. Intermediate Notation for Provenance and Workflow Reproducibility. In: IPAW. McLean, USA: Springer, 2016. p. 83–94.
- MICHENER, William; VIEGLAIS, Dave; VISION, Todd; KUNZE, John; CRUSE, Patricia; JANÉE, Greg. DataONE: Data Observation Network for Earth—Preserving data and enabling innovation in the biological and environmental sciences. **D-Lib Magazine**, Corporation for National Research Initiatives, v. 17, n. 1/2, p. 12, 2011.
- MICROSOFT. **Naming Files, Paths, and Namespaces**. Accessed: 2019-10-01. 2018. Available from: <<https://docs.microsoft.com/en-us/windows/desktop/FileIO/naming-a-file>>.
- MILES, Simon; GROTH, Paul; BRANCO, Miguel; MOREAU, Luc. The requirements of using provenance in e-science experiments. **Journal of Grid Computing**, Springer, v. 5, n. 1, p. 1–25, 2007.
- MILES, Simon; GROTH, Paul; MUNROE, Steve; MOREAU, Luc. PrIME: A methodology for developing provenance-aware applications. **ACM Transactions on Software Engineering and Methodology**, ACM, v. 20, n. 3, p. 8, 2011.
- MISSIER, Paolo; DEY, Saumen; BELHAJJAME, Khalid; CUEVAS-VICENTTÍN, Víctor; LUDÄSCHER, Bertram. D-PROV: Extending the PROV Provenance Model with Workflow Structure. In: TAPP. Lombard, USA: USENIX, 2013a. p. 1–7.
- MISSIER, Paolo; MOREAU, Luc; CHENEY, James; LEBO, Timothy; SOILAND-REYES, Stian. **PROV-Dictionary: Modeling Provenance for Dictionary Data Structures**. 2013b. Available from: <<https://www.w3.org/TR/prov-dictionary/>>.
- MOREAU, Luc; CLIFFORD, Ben; FREIRE, Juliana; FUTRELLE, Joe; GIL, Yolanda; GROTH, Paul; KWASNIKOWSKA, Natalia; MILES, Simon; MISSIER, Paolo; MYERS, Jim; PLALE, Beth; SIMMHAN, Yogesh; STEPHAN, Eric; DEN BUSSCHE, Jan Van. The open provenance model core specification (v1. 1). **Future Generation Computer Systems**, Elsevier, v. 27, n. 6, p. 743–756, 2011.
- MOREAU, Luc; FREIRE, Juliana; FUTRELLE, Joe; MCGRATH, Robert E.; MYERS, Jim; PAULSON, Patrick. The Open Provenance Model: An Overview. In: IPAW. Salt Lake City, USA: Springer Berlin Heidelberg, 2008a. v. 5272, p. 323–326. ISBN 978-3-540-89965-5.

- MOREAU, Luc; LUDÄSCHER, Bertram; ALTINTAS, Ilkay; BARGA, Roger S.; BOWERS, Shawn; CALLAHAN, Steven; CHIN JR., George; CLIFFORD, Ben; COHEN, Shirley; COHEN-BOULAKIA, Sarah; DAVIDSON, Susan; DEELMAN, Ewa; DIGIAMPIETRI, Luciano; FOSTER, Ian; FREIRE, Juliana; FREW, James; FUTRELLE, Joe; GIBSON, Tara; GIL, Yolanda; GOBLE, Carole; GOLBECK, Jennifer; GROTH, Paul; HOLLAND, David A.; JIANG, Sheng; KIM, Jihie; KOOP, David; KRENEK, Ales; MCPHILLIPS, Timothy; MEHTA, Gaurang; MILES, Simon; METZGER, Dominic; MUNROE, Steve; MYERS, Jim; PLALE, Beth; PODHORSZKI, Norbert; RATNAKAR, Varun; SANTOS, Emanuele; SCHEIDEGGER, Carlos; SCHUCHARDT, Karen; SELTZER, Margo; SIMMHAN, Yogesh L.; SILVA, Claudio; SLAUGHTER, Peter; STEPHAN, Eric; STEVENS, Robert; TURI, Daniele; VO, Huy; WILDE, Mike; ZHAO, Jun; ZHAO, Yong. Special issue: The first provenance challenge. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 20, n. 5, p. 409–418, 2008b.
- MOREAU, Luc; MISSIER, Paolo. **PROV-DM: The PROV Data Model**. 2012. Available from: <<http://www.w3.org/TR/prov-dm>>.
- MUELLER, Alexander. **5 reasons why jupyter notebooks suck**. Accessed: 2019-10-01. 2018. Available from: <<https://towardsdatascience.com/5-reasons-why-jupyter-notebooks-suck-4dc201e27086>>.
- MUNAIAH, Nuthan; KROH, Steven; CABREY, Craig; NAGAPPAN, Meiyappan. Curating GitHub for engineered software projects. **Empirical Software Engineering**, Springer, v. 22, n. 6, p. 3219–3253, 2017.
- MUNISWAMY-REDDY, Kiran-Kumar; HOLLAND, David A; BRAUN, Uri; SELTZER, Margo I. Provenance-Aware Storage Systems. In: ATC. Boston, USA: USENIX Association, 2006. p. 43–56.
- MURTA, Leonardo; BRAGANHOLO, Vanessa; CHIRIGATI, Fernando; KOOP, David; FREIRE, Juliana. noWorkflow: capturing and analyzing provenance of scripts. In: IPAW. Cologne, Germany: Springer, 2014. p. 71–83.
- MWEBAZE, Johnson; BOXHOORN, Danny; VALENTIJN, Edwin. Astro-wise: Tracing and using lineage for scientific data processing. In: NBIS. Indianapolis, USA: IEEE, 2009. p. 475–480.
- MWEBAZE, Johnson; BOXHOORN, Danny; VALENTIJN, Edwin. Dynamic Pipeline Changes in Scientific Data Processing. In: ESON. Stockholm, Sweden: IEEE, 2011. p. 263–270.

MYERS, Glenford J; BADGETT, Tom; THOMAS, Todd M; SANDLER, Corey. **The art of software testing**. Hoboken, USA: Wiley Online Library, 2004. v. 2.

NAMAKI, Mohammad Hossein; FLORATOU, Avriia; PSALLIDAS, Fotis; KRISHNAN, Subru; AGRAWAL, Ashvin; WU, Yinghui; ZHU, Yiwen; WEIMER, Markus. Vamsa: Automated Provenance Tracking in Data Science Scripts. In: KDD. Virtual Event, CA, USA: ACM, 2020. p. 1542–1551.

NEGLECTOS. **A Preliminary Analysis on the Use of Python Notebooks**. Accessed: 2019-10-01. 2018. Available from:

<<https://blog.bitergia.com/2018/04/02/a-preliminary-analysis-on-the-use-of-python-notebooks/>>.

OLIVEIRA, Daniel de; OGASAWARA, Eduardo; BAIÃO, Fernanda; MATTOSO, Marta. Scicumulus: A lightweight cloud middleware to explore many task computing paradigm in scientific workflows. In: CLOUD. Miami, USA: IEEE, 2010. p. 378–385.

OLIVEIRA, Wellington; OLIVEIRA, Daniel De; BRAGANHOLO, Vanessa. Provenance Analytics for Workflow-Based Computational Experiments: A Survey. **ACM Computing Surveys**, ACM, v. 51, n. 3, p. 53, 2018.

OUSTERHOUT, John K. Scripting: Higher level programming for the 21st century. **Computer**, IEEE, v. 31, n. 3, p. 23–30, 1998.

OXVIG, Christian Schou; ARILDSEN, Thomas; LARSEN, Torben. Storing Reproducible Results from Computational Experiments using Scientific Python Packages. In: SCIPY. Austin, USA: SciPy, 2016. p. 45–50.

PARENTE, Peter. **nbestimate**. Accessed: 2020-12-03. 2020. Available from:

<<https://nbviewer.jupyter.org/github/parente/nbestimate/blob/master/estimate.ipynb>>.

PATTERSON, Evan. **The algebra and machine representation of statistical models**. 2020. PhD thesis – Department of Statistics, Stanford University, CA, USA.

PATTERSON, Evan; BALDINI, Ioana; MOJSILOVIC, Aleksandra; VARSHNEY, Kush R. Machine Representation of Data Analyses: Towards a Platform for Collaborative Data Science. In: AAAI. Palo Alto, USA: AAAI Press, 2017a.

PATTERSON, Evan; BALDINI, Ioana; MOJSILOVIC, Aleksandra; VARSHNEY, Kush R. Semantic Representation of Data Science Programs. In: IJCAI. Stockholm, Sweden: ijcai.org, 2018. p. 5847–5849.

PATTERSON, Evan; MCBURNEY, Robert; SCHMIDT, Holly; BALDINI, Ioana; MOJSILOVIĆ, Aleksandra; VARSHNEY, Kush R. Dataflow representation of data analyses: Toward a platform for collaborative data science. **IBM Journal of Research and Development**, IBM, v. 61, n. 6, p. 9–1, 2017b.

PAWLIK, Mateusz; AUGSTEN, Nikolaus. Tree edit distance: Robust and memory-efficient. **Information Systems**, Elsevier, v. 56, p. 157–173, 2016.

PÉREZ, Fernando; GRANGER, Brian E. IPython: a system for interactive scientific computing. **Computing in Science & Engineering**, IEEE, v. 9, n. 3, p. 21–29, 2007.

PETERSEN, Kai; FELDT, Robert; MUJTABA, Shahid; MATTSSON, Michael. Systematic Mapping Studies in Software Engineering. In: EASE. University of Bari, Italy: ACM, 2008. v. 8, p. 68–77.

PETRICEK, Tomas; GEDDES, James; SUTTON, Charles. Wrattler: Reproducible, live and polyglot notebooks. In: TAPP. London, UK: USENIX, 2018.

PIMENTEL, João Felipe. **Prov Survey GitHub page**. Accessed: 2021-02-19. Jan. 2017. Available from: <<https://github.com/JoaoFelipe/prov-survey>>.

PIMENTEL, João Felipe; BRAGANHOLO, Vanessa; MURTA, Leonardo; FREIRE, Juliana. Collecting and analyzing provenance on interactive notebooks: when IPython meets noWorkflow. In: TAPP. Edinburgh, Scotland: USENIX, 2015. p. 1–6.

PIMENTEL, João Felipe; DEY, Saumen; MCPHILLIPS, Timothy; BELHAJJAME, Khalid; KOOP, David; MURTA, Leonardo; BRAGANHOLO, Vanessa; LUDÄSCHER, Bertram. Yin & Yang: demonstrating complementary provenance from noWorkflow & YesWorkflow. In: IPAW. McLean, USA: Springer, 2016a. p. 161–165.

PIMENTEL, João Felipe; FREIRE, Juliana; BRAGANHOLO, Vanessa; MURTA, Leonardo. Tracking and analyzing the evolution of provenance from scripts. In: IPAW. McLean, USA: Springer, 2016b. p. 16–28.

PIMENTEL, João Felipe; FREIRE, Juliana; MURTA, Leonardo; BRAGANHOLO, Vanessa. A survey on collecting, managing, and analyzing provenance from scripts. **ACM Computing Surveys**, ACM, v. 52, n. 3, p. 1–38, 2019a.

PIMENTEL, João Felipe; FREIRE, Juliana; MURTA, Leonardo; BRAGANHOLO, Vanessa. Fine-grained provenance collection over scripts through program slicing. In: IPAW. McLean, USA: Springer, 2016c. p. 199–203.

- PIMENTEL, João Felipe; MISSIER, Paolo; MURTA, Leonardo; BRAGANHOLO, Vanessa. **Versioned-PROV**. en. Accessed: 2018-03-11. 2018a. Available from: [<https://dew-uff.github.io/versioned-prov/>](https://dew-uff.github.io/versioned-prov/).
- PIMENTEL, João Felipe; MISSIER, Paolo; MURTA, Leonardo; BRAGANHOLO, Vanessa. Versioned-PROV: A PROV Extension to Support Mutable Data Entities. In: IPAW. London, UK: Springer, 2018b. p. 87–100.
- PIMENTEL, João Felipe; MURTA, Leonardo; BRAGANHOLO, Vanessa; FREIRE, Juliana. A large-scale study about quality and reproducibility of jupyter notebooks. In: MSR. Montreal, Canada: IEEE, 2019b. p. 507–517.
- PIMENTEL, João Felipe; MURTA, Leonardo; BRAGANHOLO, Vanessa; FREIRE, Juliana. noWorkflow: a tool for collecting, analyzing, and managing provenance from python scripts. **Proceedings of the VLDB Endowment**, v. 10, n. 12, p. 1841–1844, 2017.
- PIMENTEL, João Felipe; MURTA, Leonardo; BRAGANHOLO, Vanessa; FREIRE, Juliana. Understanding and Improving the Quality and Reproducibility of Jupyter Notebooks. **Empirical Software Engineering**. In press, p. 1–63, 2021.
- PINTO, Gustavo; WIESE, Igor; DIAS, Luiz Felipe. How do scientists develop scientific software? an external replication. In: SANER. Campobasso, Italy: IEEE, 2018. p. 582–591.
- POMOGAJKO, Kirill. **Why I Don't Like Jupyter (FKA IPython Notebook)**. Accessed: 2019-10-01. 2015. Available from: [<https://yihui.name/en/2018/09/notebook-war/>](https://yihui.name/en/2018/09/notebook-war/).
- PONTES, Vynicius Morais. **Reducing the Storage Overhead of the noWorkflow Content Database by using Git**. 2018. s. 17. Bachelor's Thesis – Universidade Federal Fluminense.
- PORGES, Arthur. A set of eight numbers. **The American Mathematical Monthly**, Taylor & Francis, v. 52, n. 7, p. 379–382, 1945.
- PRABHU, Prakash; KIM, Hanjun; OH, Taewook; JABLIN, Thomas B.; JOHNSON, Nick P.; ZOUFALY, Matthew; RAMAN, Arun; LIU, Feng; WALKER, David; ZHANG, Yun; GHOSH, Soumyadeep; AUGUST, David I.; HUANG, Jialu; BEARD, Stephen. A survey of the practice of computational science. In: SC. Seattle, USA: ACM, 2011. p. 1–12.
- PYPA. **Python Packaging Documentation: install\_requires vs requirements files**. Accessed: 2020-10-08. 2020. Available from: [<https://packaging.python.org/discussions/install-requires-vs-requirements>](https://packaging.python.org/discussions/install-requires-vs-requirements/).

- PYTHON-WIKI. **Python Testing Tools Taxonomy**. Accessed: 2019-10-01. 2019. Available from: <<https://wiki.python.org/moin/PythonTestingToolsTaxonomy>>.
- RAMAKRISHNAN, Arun; SINGH, Gurmeet; ZHAO, Henan; DEELMAN, Ewa; SAKELLARIOU, Rizos; VAHI, Karan; BLACKBURN, Kent; MEYERS, David; SAMIDI, Michael. Scheduling data-intensiveworkflows onto storage-constrained distributed resources. In: CCGRID. Rio de Janeiro, Brazil: IEEE, 2007. p. 401–409.
- RAMAKRISHNAN, Raghu; ULLMAN, Jeffrey D. A survey of deductive database systems. **The Journal of Logic Programming**, Elsevier, v. 23, n. 2, p. 125–149, 1995. ISSN 0743-1066.
- RAMASUBRAMANI, Vyas; ADORF, Carl; DODD, Paul; DICE, Bradley; GLOTZER, Sharon. signac: A Python framework for data and workflow management. In: SCIPY. Austin, USA: SciPy Conference, 2018.
- REPROZIP. **Making Jupyter Notebooks Reproducible with ReproZip**. Accessed: 2019-10-01. 2017. Available from: <<https://docs.reprozip.org/en/1.0.x/jupyter.html>>.
- ROSSUM, Guido van; WARSAW, Barry; COGHLAN, Nick. **PEP 8: style guide for Python code**. Accessed: 2019-10-01. 2001.
- RUGGIERI, Salvatore; PEDRESCHI, Dino; TURINI, Franco. Data Mining for Discrimination Discovery. **ACM Trans. Knowl. Discov. Data**, ACM, New York, USA, v. 4, n. 2, 9:1–9:40, May 2010. ISSN 1556-4681. DOI: 10.1145/1754428.1754432. Available from: <<http://doi.acm.org/10.1145/1754428.1754432>>.
- RULE, Adam; TABARD, Aurélien; HOLLAN, James D. Exploration and Explanation in Computational Notebooks. In: CHI. Montreal QC, Canada: ACM, 2018. 32:1–32:12. ISBN 978-1-4503-5620-6. DOI: 10.1145/3173574.3173606.
- RUNNALLS, Andrew R. Aspects of CXXR internals. **Computational Statistics**, Springer, v. 26, n. 3, p. 427–442, 2011.
- RUNNALLS, Andrew R; SILLES, Chris A. CXXR: An ideas hatchery for future R development. In: JSM. Miama Beach, USA: AMSTAT, 2011. p. 1–9.
- RUNNALLS, Andrew; SILLES, Chris. Provenance tracking in R. In: IPAW. Santa Barbara, USA: Springer, 2012. p. 237–239.
- SAMUEL, Sheeba; KÖNIG-RIES, Birgitta. ProvBook: Provenance-based Semantic Enrichment of Interactive Notebooks for Reproducibility. In: ISWC. Monterey, USA: CEUR-WS.org, 2018.

- SHAPIRO, Samuel Sanford; WILK, Martin B. An analysis of variance test for normality (complete samples). **Biometrika**, JSTOR, v. 52, n. 3/4, p. 591–611, 1965.
- SHEN, Helen et al. Interactive notebooks: Sharing the code. **Nature**, Macmillan Publishers Ltd., London, England, v. 515, n. 7525, p. 151–152, 2014.
- SILLES, Chris A; RUNNALLS, Andrew R. Provenance-awareness in R. In: IPAW. Troy, USA: Springer, 2010. p. 64–72.
- SILLES, Christopher Anthony. **Provenance-aware CXXR**. 2014. PhD thesis – University of Kent.
- SILVA, Cláudio T; TOHLIN, Joel E. Computational Provenance. **Computing in Science & Engineering**, v. 10, n. 3, p. 9–10, 2008.
- SILVA, Vítor; CAMPOS, Vinícius; GUEDES, Thaylon; CAMATA, José; OLIVEIRA, Daniel de; COUTINHO, Alvaro LGA; VALDURIEZ, Patrick; MATTOSO, Marta. DfAnalyzer: Runtime dataflow analysis tool for Computational Science and Engineering applications. **SoftwareX**, Elsevier, v. 12, p. 100592, 2020.
- SILVA, Vítor; OLIVEIRA, Daniel de; VALDURIEZ, Patrick; MATTOSO, Marta. DfAnalyzer: runtime dataflow analysis of scientific applications using provenance. **Very Large Data Bases**, Proceedings of the VLDB Endowment, v. 11, n. 12, p. 2082–2085, 2018a.
- SILVA, Vítor; SOUZA, Renan; CAMATA, Jose; OLIVEIRA, Daniel de; VALDURIEZ, Patrick; COUTINHO, Alvaro LGA; MATTOSO, Marta. Capturing provenance for runtime data analysis in computational science and engineering applications. In: IPAW. London, UK: Springer, 2018b. p. 183–187.
- SIMMHAN, Yogesh L; PLALE, Beth; GANNON, Dennis. A survey of data provenance in e-science. **SIGMOD Record**, ACM, v. 34, n. 3, p. 31–36, 2005a.
- SIMMHAN, Yogesh L; PLALE, Beth; GANNON, Dennis. **A Survey of Data Provenance Techniques**. v. 47405. Indiana University, Bloomington IN, 2005b.
- SIMMHAN, Yogesh L; PLALE, Beth; GANNON, Dennis. Karma2: Provenance management for data-driven workflows. **International Journal of Web Services Research**, IGI Global, v. 5, n. 2, p. 1–22, 2008.
- SORLIN, Sébastien; SOLNON, Christine. Reactive tabu search for measuring graph similarity. In: IAPR. Poitiers, France: Springer, 2005. p. 172–182.
- SOUZA, Renan; MATTOSO, Marta. Provenance of dynamic adaptations in user-steered dataflows. In: IPAW. London, UK: Springer, 2018. p. 16–29.

- STALEY, Tim. **Making Git and Jupyter Notebooks play nice**. Accessed: 2019-10-01. 2017. Available from: <<http://timstaley.co.uk/posts/making-git-and-jupyter-notebooks-play-nice/>>.
- STAMATOIANNAKIS, Manolis; GROTH, Paul; BOS, Herbert. Looking inside the black-box: capturing data provenance using dynamic instrumentation. In: IPAW. Cologne, Germany: Springer, 2014. p. 155–167.
- STEVENS, Jean-Luc Richard; ELVER, Marco; BEDNAR, James A. An automated and reproducible workflow for running and analyzing neural simulations using Lancet and IPython Notebook. **Frontiers in Neuroinformatics**, Frontiers, v. 7, n. 44, p. 44, 2013.
- STONEBRAKER, Michael; HELD, Gerald; WONG, Eugene; KREPS, Peter. The design and implementation of INGRES. **ACM Transactions on Database Systems**, ACM, v. 1, n. 3, p. 189–222, 1976.
- STRACHEY, Christopher. Fundamental concepts in programming languages. **Higher-order and symbolic computation**, Springer, v. 13, n. 1-2, p. 11–49, 2000.
- STUDENT. The probable error of a mean. **Biometrika**, JSTOR, p. 1–25, 1908.
- SUBRAMANIAN, Krishna; MAAS, Johannes; BORCHERS, Jan. TRACTUS: Understanding and Supporting Source Code Experimentation in Hypothesis-Driven Data Science. In: CHI. Honolulu, USA: ACM, 2020. p. 1–12.
- TAN, Wang Chiew et al. Provenance in Databases: Past, Current, and Future. **IEEE Data Engineering Bulletin**, v. 30, n. 4, p. 3–12, 2007.
- TARIQ, Dawood; ALI, Maisem; GEHANI, Ashish. Towards Automated Collection of Application-Level Data Provenance. In: TAPP. Boston, USA: USENIX, 2012. p. 1–5.
- TIM; DOORKNOB. **Is space not allowed in a filename?** Accessed: 2019-10-01. 2014. Available from: <<https://unix.stackexchange.com/q/148043>>.
- TRAVASSOS, Guilherme Horta; BARROS, Márcio O. Contributions of in virtuo and in silico experiments for the future of empirical studies in software engineering. In: WSESE. Rome, Italy: IEEE, 2003. p. 117–130.
- TUFANO, Michele; PALOMBA, Fabio; BAVOTA, Gabriele; DI PENTA, Massimiliano; OLIVETO, Rocco; DE LUCIA, Andrea; POSHYVANYK, Denys. There and back again: Can you compile that snapshot? **Journal of Software: Evolution and Process**, Wiley Online Library, v. 29, n. 4, e1838, 2017.
- VALEUR, Håvar. **Tracking the lineage of arbitrary processing sequences**. 2005. s. 91. PhD thesis – Norwegian University of Science and Technology, Trondheim.

- VAN DER HOEK, André. Design-time product line architectures for any-time variability. **Science of Computer Programming**, Elsevier, v. 53, n. 3, p. 285–304, 2004.
- VERNOOJJ, Jelmer. **Dulwich**. Accessed: 2021-01-07. 2018. Available from: <https://github.com/dulwich/dulwich>.
- WALKER, Edward; GUIANG, Chona. Challenges in executing large parameter sweep studies across widely distributed computing environments. In: CLADE. Monterey Bay, USA: ACM, 2007. p. 11–18.
- WANG, Gao; PENG, Bo. Script of Scripts: A pragmatic workflow system for daily computational research. **PLOS Computational Biology**, Public Library of Science, v. 15, n. 2, e1006843, 2019.
- WANG, Jianwu; CRAWL, Daniel; PURAWAT, Shweta; NGUYEN, Mai; ALTINTAS, Ilkay. Big data provenance: Challenges, state of the art and opportunities. In: BIGDATA. Santa Clara, USA: IEEE, 2015. p. 2509–2516.
- WANG, Jiawei; LI, Li; ZELLER, Andreas. Better code, better sharing: on the need of analyzing jupyter notebooks. In: ICSE-NIER. Seoul, South Korea: ACM, 2020. (ICSE), p. 53–56.
- WANG, Jiawei; TZU-YANG, KUO; LI, Li; ZELLER, Andreas. Assessing and Restoring Reproducibility of Jupyter Notebooks. In: ASE. Melbourne, Australia: IEEE/ACM, 2020. p. 138–149.
- WENSKOVITCH, John; ZHAO, Jian; CARTER, Scott; COOPER, Matthew; NORTH, Chris. Albireo: An Interactive Tool for Visually Summarizing Computational Notebook Structure. In: VDS. Vancouver, BC, Canada: IEEE, 2019. p. 1–10.
- WICKERT, Andrew D. Open-source modular solutions for flexural isostasy: gFlex v1. 0. **Geoscientific Model Development**, Copernicus GmbH, v. 9, n. 3, p. 997–1017, 2016.
- WILCOXON, Frank. Individual comparisons by ranking methods. In: BREAKTHROUGHS in statistics. New York, USA: Springer, 1992. p. 196–202.
- WILSON, Greg; ARULIAH, D. A.; BROWN, C. Titus; CHUE HONG, Neil P.; DAVIS, Matt; GUY, Richard T.; HADDOCK, Steven H. D.; HUFF, Kathryn D.; MITCHELL, Ian M.; PLUMBLEY, Mark D.; WAUGH, Ben; WHITE, Ethan P.; WILSON, Paul. Best Practices for Scientific Computing. **PLOS Biology**, Public Library of Science, v. 12, n. 1, p. 1–7, Jan. 2014. DOI: 10.1371/journal.pbio.1001745.

- WINCKEL, Greg von. **The quantum harmonic oscillator with two electrons**. Accessed: 2018-06-15. 2014. Available from: <[scientificpython.net/pyblog/quantum-harmonic-oscillator-with-two-interacting-electrons](http://scientificpython.net/pyblog/quantum-harmonic-oscillator-with-two-interacting-electrons)>.
- WOHLIN, Claes. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In: 38. EASE. London, UK: ACM, 2014. ISBN 978-1-4503-2476-2.
- WOLSTENCROFT, Katherine; HAINES, Robert; FELLOWS, Donal; WILLIAMS, Alan; WITHERS, David; OWEN, Stuart; SOILAND-REYES, Stian; DUNLOP, Ian; NENADIC, Aleksandra; FISHER, Paul, et al. The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. **Nucleic Acids Research**, Oxford Univ Press, W557, n. 61, w557–w561, 2013.
- XU, Zhaogui; QIAN, Ju; CHEN, Lin; CHEN, Zhifei; XU, Baowen. Static Slicing for Python First-Class Objects. In: QSIC. Nanjing, China: IEEE, 2013. p. 117–124.
- YSEARKA. **importing seaborn changes matplotlib graphs**. Accessed: 2021-01-26. 2016. Available from: <<https://stackoverflow.com/questions/38077854/importing-seaborn-changes-matplotlib-graphs?noredirect=1&lq=1>>.
- ZANIOLO, Carlo. The database language GEM. **SIGMOD Record**, ACM, v. 13, n. 4, p. 207–218, 1983.
- ZHANG, Qian; CAO, Yang; WANG, Qiwen; VU, Duc; THAVASIMANI, Priyaa; MCPHILLIPS, Timothy M; MISSIER, Paolo; SLAUGHTER, Peter; JONES, Christopher; JONES, Matthew B, et al. Revealing the Detailed Lineage of Script Outputs Using Hybrid Provenance. **International Journal of Digital Curation**, v. 12, n. 2, p. 390–408, 2017.
- ZHANG, Yi; IVES, Zachary G. Finding related tables in data lakes for interactive data science. In: SIGMOD. Portland, USA: ACM, 2020. p. 1951–1966.
- ZHANG, Yi; IVES, Zachary G. Juneau: data lake management for Jupyter. **Proceedings of the VLDB Endowment**, v. 12, n. 12, 2019.
- ZHAO, Yong; HATEGAN, Mihael; CLIFFORD, Ben; FOSTER, Ian; VON LASZEWSKI, Gregor; NEFEDOVA, Veronika; RAICU, Ioan; STEF-PRAUN, Tiberiu; WILDE, Michael. Swift: Fast, reliable, loosely coupled parallel computation. In: SERVICES. Salt Lake City, USA: IEEE, 2007. p. 199–206.
- ZHAO, Yong; LU, Shiyong. A logic programming approach to scientific workflow provenance querying. In: IPAW. Salt-Lake City, USA: Springer, 2008. p. 31–44.

---

ZHAO, Yong; WILDE, Michael; FOSTER, Ian. Applying the virtual data provenance model. In: IPAW. Chicago, Illinois, USA: Springer, 2006. p. 148–161.

## APPENDIX A - Versioned-PROV

### A.1 Introduction

This appendix describes Versioned-PROV, a PROV extension to support mutable data entities (PIMENTEL et al., 2018b). Versioned-PROV was proposed in collaboration with Paolo Missier (Newcastle University), Leonardo Murta (UFF), and Vanessa Braganholo (UFF).

As presented in Chapter 4, PROV (and its predecessor, OPM (MOREAU et al., 2008a)) has been applied to describe the provenance gathered from workflow systems (COSTA et al., 2013), operating systems (MUNISWAMY-REDDY et al., 2006), and scripts (ANGELINO; YAMINS; SELTZER, 2010). Tools that collect operating system provenance map users as agents, file objects and program arguments as entities, and program executions and system calls as activities (MUNISWAMY-REDDY et al., 2006). Workflow systems map data as entities and processing steps as activities (COSTA et al., 2013). Finally, tools that collect coarse-grained provenance from scripts map data in function arguments and data values obtained from return statements as entities, and function calls as activities (ANGELINO; YAMINS; SELTZER, 2010).

In the aforementioned approaches, entities are immutable data that go through processing steps (modeled as activities) to produce new immutable data (modeled as entities). The assumption of immutable entities also exists in the PROV data model, where changes to an entity  $e$  are explicitly represented by the creation of a new entity  $e'$  generated by the activities that use the original  $e$ .

No known approaches use PROV to describe fine-grained provenance from scripts, with support for variables and mutable data structures. The goal of this appendix is to extend the well-known concepts of coarse-grained provenance for scripts, which is limited to function arguments and function calls, to (1) script variables, (2) expressions with operators, and (3) assignments, thus realizing fine-grained provenance for scripts. Specifically, we note that we can map script variables to entities, expressions with operators to activities that generate new entities, and assignments to activities that produce derivations, i.e., from expression results to

variables. For example,  $a = b + c$  can be mapped as an activity `+` that uses the entities `b` and `c` to generate the derived entity `sum`, and an assignment activity that uses `sum` to generate the derived entity `a`.

This is a challenging goal because using PROV to represent fine-grained provenance suffers from two main problems: (P1) when an entity that represents a collection is changed (e.g., a list is updated to add an element), a new entity should be created, together with multiple new relationships, connecting the new entity to each of the existing or new entities that represent the elements of the collection; and (P2) when more than one variable is assigned to the same collection, and one of the variables changes, all other variables should also change, as they refer to the same memory area. This means that a new entity should be created for each variable that contains the collection, together with edges for all entities that represent the elements of the collection. As we show in Section A.4, these problems lead to  $O(N)$  and  $\Omega(R \times N)$  extra elements in the provenance graph, respectively, for collections with  $N$  elements and  $R$  references.

PROV has been extended in many different ways (COSTA et al., 2013; GARIJO; GIL, 2012; MISSIER et al., 2013a), but most extensions focus only on representing domain-specific provenance and do not improve the support for data structures. The PROV-Dictionary extension (MISSIER et al., 2013b) improves the support for data structures in PROV by adding derivation statements that indicate that a new collection shares most elements of the old one, but with the insertion or removal of specific elements. This solves P1 since it reduces the number of edges to 1. However, it still suffers from P2, since it requires updating all entities that refer to the same collection when it changes, which leads to  $\Omega(R)$  extra elements.

In this appendix, we present Versioned-PROV, an extension that adds *reference sharing* and *checkpoints* to PROV. Checkpoints solve problem P1 in  $O(1)$  by allowing the representation of multiple versions of collections with a single entity. Reference sharing solves problem P2 in  $O(1)$  by allowing collections to be represented only once and referred to by other entities through reference derivations plus checkpoints to indicate states.

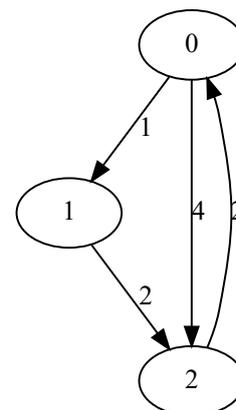
This appendix is organized as follows. Section A.2 presents a running example, which is based on the *Floyd-Warshall* algorithm (FLOYD, 1962). Section A.3 introduces Versioned-PROV. Section A.4 evaluates the approach by comparing it to PROV and PROV-Dictionary. Section A.5 concludes the appendix.

```

1  m = 10000 # max value
2  result = dist = [
3      [0, 1, 4],
4      [m, 0, 2],
5      [2, m, 0]]
6  nodes = len(dist)
7  indexes = range(nodes)
8  for k in indexes:
9      distk = dist[k]
10     for i in indexes:
11         if i == k: continue
12         disti = dist[i]
13         for j in indexes:
14             if j == i or j == k: continue
15             ikj = disti[k] + distk[j]
16             if disti[j] > ikj:
17                 disti[j] = ikj
18  print(result[0][2])

```

(A)



(B)

Figure A.1: *Floyd-Warshall* implementation (A) and encoded input graph (B).

## A.2 Running Example

While Versioned-PROV intends to be generic enough for any situation that requires sharing references to mutable collections in PROV, we use fine-grained script provenance as a case study for presenting our extension. More specifically, we use the *Floyd-Warshall* algorithm (FLOYD, 1962) as a base to describe and evaluate the mapping of fine-grained provenance from scripts using Versioned-PROV. This algorithm has relevant applications, such as finding the shortest path between two addresses in a navigation system.

The algorithm calculates the length of the shortest path between all pairs of nodes in a weighted graph. It achieves this by updating the distance of the path from node  $i$  to node  $j$  if there is a node  $k$  for which the distance of the path from  $i$  to  $k$  plus the distance of the path from  $k$  to  $j$  is shorter than the distance from  $i$  to  $j$ . The result of *Floyd-Warshall* is the set of shortest distances among all pairs of nodes, but it does not produce the actual shortest paths. However, observing that the path between two nodes is defined by the sum of two other paths, here we show that we can use the fine-grained provenance of a given output distance to obtain the actual paths that have that distance.

Figure A.1 presents a Python implementation of *Floyd-Warshall* with a predefined input graph. Line 18 prints the distance of the shortest path from 0 to 2. While there is a direct edge with cost 4, the actual result is 3, because the shortest path goes from 0 to 1, with cost 1, and then from 1 to 2, with cost 2. After the algorithm changes the result matrix, querying the provenance of `result[0][2]` in line 18 should indicate that it derives from `result[0][1]` and `result[1][2]`.

Table A.1: Versioned-PROV types.

Type	Statement	Meaning
Reference	wasDerivedFrom	The generated entity derived from the used entity by reference, indicating that both have the same numbers.
Put	hadMember	Put a member into a collection <i>key</i> position at a given <i>checkpoint</i> . Using a placeholder as member indicates a deletion.

## A.3 Versioned-PROV

Versioned-PROV adds the concepts of checkpoints, reference sharing, and accesses to PROV. Different from plain PROV, which assumes immutable entities, a Versioned-PROV entity may represent multiple versions of a data object. We present Versioned-PROV concepts in Section A.3.1. In Section A.3.2, we detail Versioned-PROV by presenting a mapping of a part assignment in the *Floyd-Warshall* algorithm, and contrasting it to PROV and PROV-Dictionary.

### A.3.1 Concepts

The PROV data model is based on the idea of instantaneous transition events that describe usage, generation, and invalidation of entities (MOREAU; MISSIER, 2012). These events are important to describe the provenance timeline without explicit time and ordering. Versioned-PROV builds on top of PROV events and determines that a version of a data object changes on a generation event, and is accessed on a usage event. Instead of relying on the implicit ordering of events from PROV, Versioned-PROV uses *checkpoint* attributes to tag events and changes on entities. Then, it uses the explicit ordering of *checkpoints* to obtain a version of a data object. Hence, we require a total order to be defined on the set of checkpoints. Our implementation of *Floyd-Warshall* uses *timestamps* as checkpoints, but the figures in this appendix use *sequential numbers*. Both can be ordered. noWorkflow 2 uses the elapsed time since the beginning of the trial as *checkpoint*, which can also be ordered.

As an extension of PROV, Versioned-PROV follows its semantics. Thus, despite the goal of representing multiple versions of a data object, an entity in PROV can only be generated once, according to the unique-generation constraint of PROV (MOREAU; MISSIER, 2012). Thus, the only mutability on the Versioned-PROV entities occurs in the memberships of collection entities. A collection may have different members at different moments, but the operations that put and delete members from a collection are incremental. It means that if a collection  $c$  had an entity  $e_1$  at checkpoint 1 and an operation put the entity  $e_2$  into a different position of  $c$  at checkpoint 2, then  $c$  had both  $e_1$  and  $e_2$  at checkpoint 2.

Table A.2: Versioned-PROV attributes.

Attribute	Range	Statement	Meaning
checkpoint	Sortable Value	hadMember	Checkpoint of the collection update. Required for <i>hadMember</i> with type <i>Put</i> .
checkpoint	Sortable Value	Events (e.g., used, wasDerivedFrom)	Checkpoint of the event. Required for <i>wasDerivedFrom</i> with type <i>Reference</i> .
key	String	hadMember	The position of <i>Put</i> .
key	String	wasDerivedFrom	The position of the accessed <i>collection</i> entity.
collection	Entity Id	wasDerivedFrom	Collection entity that was accessed or changed.
access	'r' or 'w'	wasDerivedFrom	Indicates whether an access reads ('r') an element from a collection or writes ('w') into it.

Different from PROV and PROV-Dictionary that use copy-by-value to represent data structure assignments and derivations, Versioned-PROV uses copy-by-reference. Hence, it defines the data structure once and uses *reference sharing* to indicate that more than one entity refers to the same data structure. When generating and using Versioned-PROV entities, one must indicate a checkpoint to unfold the specific version of the data structure for any given event. When an entity associated with a data structure changes at a given checkpoint, we can infer that all entities that share reference with it also changes at the same checkpoint, without any extra explicit statements.

Versioned-PROV uses PROV optional attributes and defines types to extend PROV. Table A.1 presents the Versioned-PROV types, and Table A.2 presents the Versioned-PROV attributes. The attributes *key*, *collection*, and *access* of *wasDerivedFrom* may only be used when the derivation is related to an access or collection update. Similarly, the type *Put* can only appear in data structures, to define their items. Differently, the attribute *checkpoint* and the type *Reference* can appear anywhere, despite affecting only collection entities. This keeps the model consistent in all situations that involve using and generating entities.

### A.3.2 Mapping Example

We use the script example of Section A.2 to detail Versioned-PROV in contrast to PROV and PROV-Dictionary. We map the execution provenance of the *Floyd-Warshall* algorithm (Figure A.1) to these three approaches. Due to space constraints, we present only the first execution of the part assignment in line 17 of Figure A.1 (i.e.,  $dist_i[j] = ikj$ ). The complete mapping is available at (PIMENTEL et al., 2018a).

Figure A.2, Figure A.3, and Figure A.4 present the part assignment mapped to plain PROV, PROV-Dictionary, and Versioned-PROV, respectively. In our mappings, we name entities based

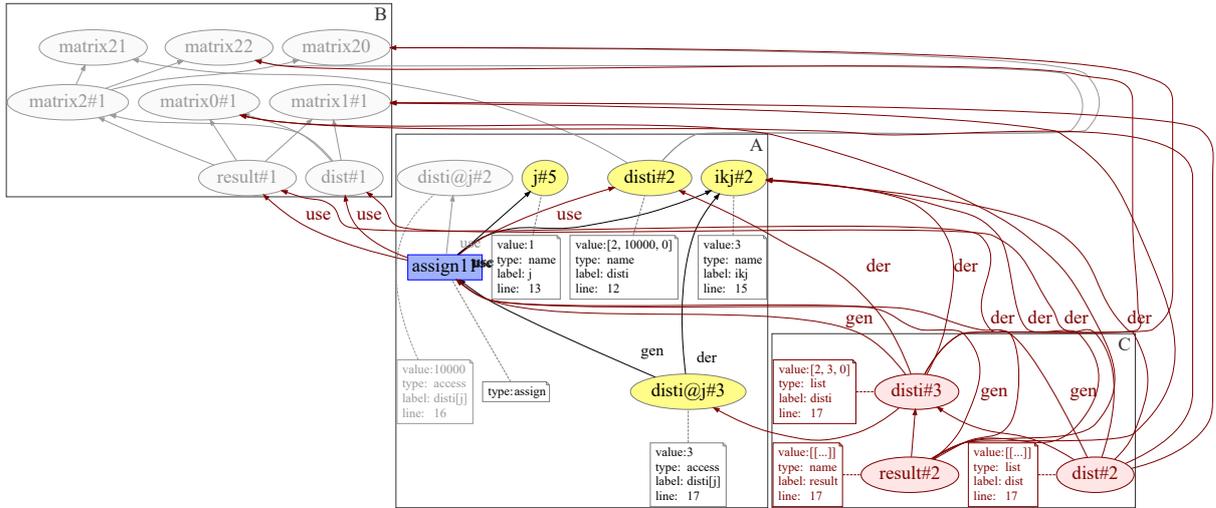


Figure A.2: Plain PROV mapping of  $dist_i[j] = ik_j$ .

on their textual representations. Since a textual element (e.g., a variable) can be represented by multiple entities, we enumerate them. Thus,  $ik_j\#2$  denotes the second entity that represents the variable  $ik_j$  (as defined in line 15 of Figure A.1). In addition to this numbering, we change the notation of accesses to avoid using escaping characters to represent square brackets. Instead, we use the collection name followed by “@” and the accessed key. For instance, we use  $dist_i@j$  to represent  $dist_i[j]$  (lines 16-17 of Figure A.1). Note in region A of these figures that we have both  $dist_i@j\#2$  in gray, representing  $dist_i[j]$  of line 16, and  $dist_i@j\#3$  in yellow, representing  $dist_i[j]$  of line 17. The latter is the result of the part assignment.

We divide these figures into three regions: **A** represents the base part assignment that exists in all approaches; **B** represents a portion of the matrix that existed before this operation; and **C** represents the overhead entities (i.e., entities that are specific to an approach) that were generated as consequence of the part assignment. Note that Figure A.4 has no region C since Versioned-PROV does not have overhead entities. All the entities that exist in Versioned-PROV also exist in the other approaches.

We also use the color red to denote the overhead. Note that plain PROV has a bigger overhead than PROV-Dictionary, which has a bigger overhead than Versioned-PROV. This occurs due to the problems P1 and P2 mentioned in the introduction. Additionally, we use gray to indicate the portion of the provenance graph that is not related to the part assignment operation. As expected, all nodes and edges in region B are gray. The only gray node outside region B is  $dist_i@j\#2$  in region A. This node appears due to the if condition in line 16 of Figure A.1. Hence, it is specific to this algorithm and not a generic node that occurs in all part assignments.

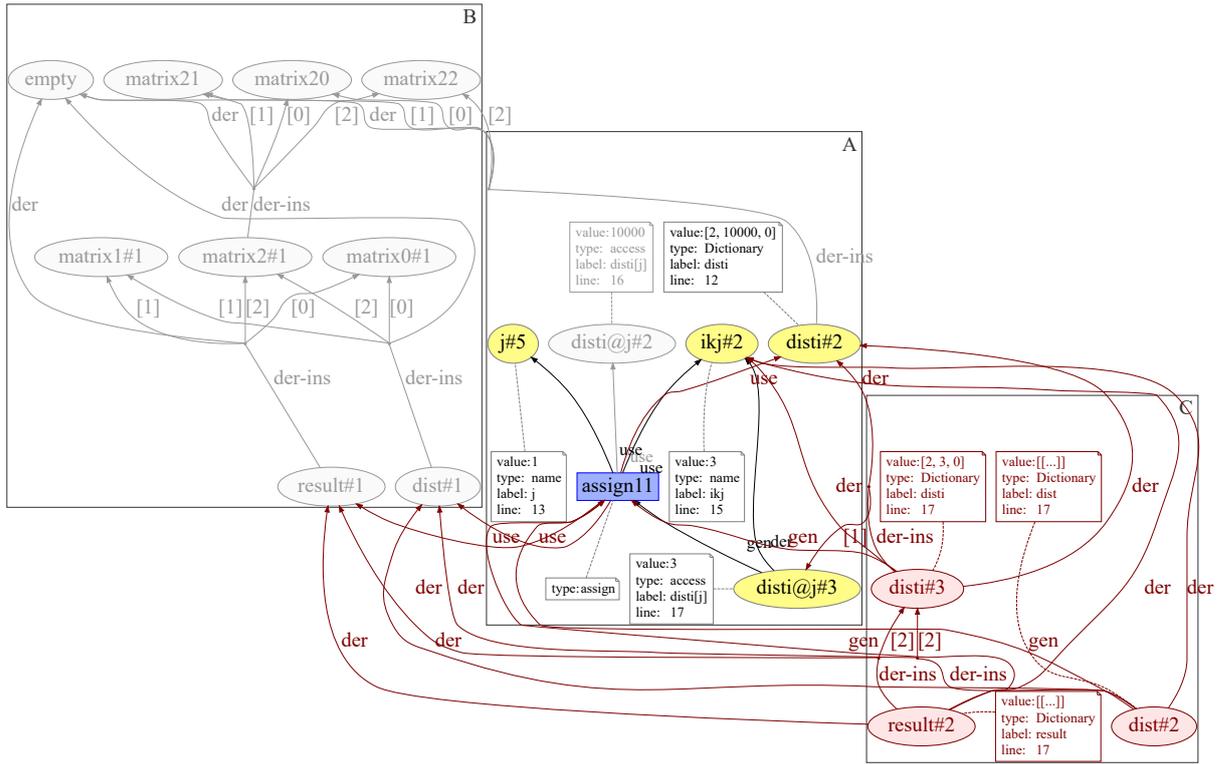


Figure A.3: PROV-Dictionary mapping of  $dist i [ j ] = i k j$ .

The operation  $dist i [ j ] = i k j$  is putting the value of  $i k j$  into the position  $j$  of  $dist i$ . In region A of all figures,  $i k j \# 2$  represents the variable  $i k j$ ;  $j \# 5$  represents  $j$ ; and  $dist i \# 2$  represents  $dist i$ . Additionally,  $dist i @ j \# 3$  represents the resulting  $dist i [ j ]$ . Note that  $dist i$  in this execution is the same list as  $dist [ 2 ]$ , represented by the entity  $matrix2 \# 1$  (i.e., they point to the same memory area). Note also that  $dist$  and  $result$  are the same matrix.

Since entities are immutable in PROV and PROV-Dictionary, an update in a collection ( $dist i \# 2$  in region A) requires the creation of a new collection ( $dist i \# 3$  in region C) that contains the updated members. PROV suffers from P1, thus it reconstructs the membership of the new entity by using  $N$  *hadMember* relationships in a collection with  $N$  members (3 in this case). We represent these relationships by edges without labels in Figure A.2. PROV-Dictionary, on the other hand, uses a single *derivedByInsertionFrom* (*der-ins* edges in Figure A.3) to indicate that a collection was updated by the insertion of a member at a position ( $dist i \# 3$  derived from  $dist i \# 2$  by the insertion of  $dist i @ j \# 3$  from region A at position 1).

As stated before,  $dist i \# 2$  represents the same value as  $matrix2 \# 1$ . Thus, we would have to update  $matrix2 \# 1$  to reflect the change. This does not occur because  $matrix2 \# 1$

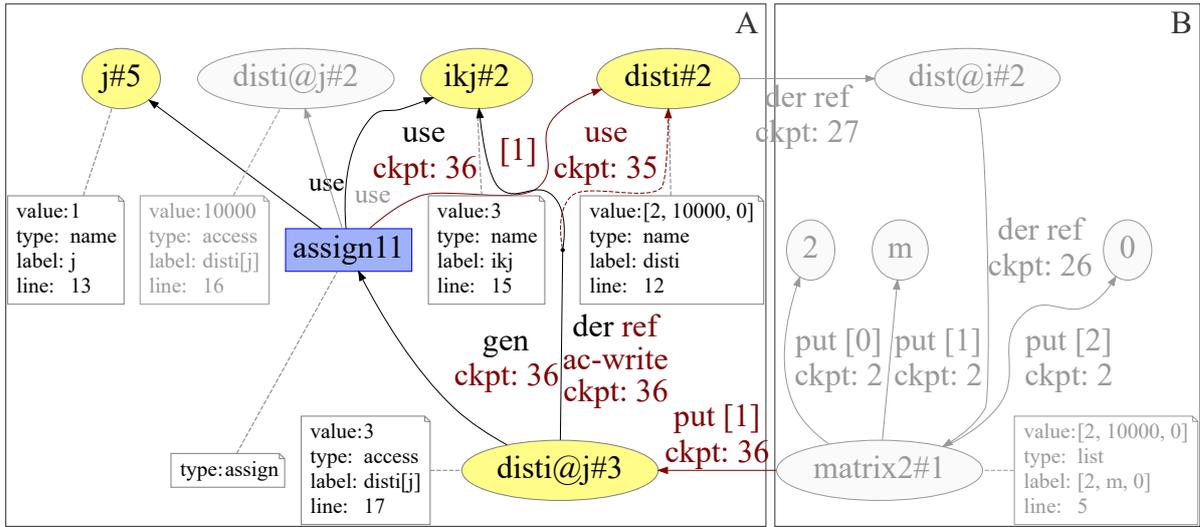


Figure A.4: Versioned-PROV mapping of  $\text{dist } i[j] = ikj$ .

is out of the scope of the execution at this point and cannot be directly used without an access to  $\text{dist}\#1$  or  $\text{result}\#1$ . Due to P2, plain PROV and PROV-Dictionary update  $\text{dist}\#1$  and  $\text{result}\#1$  by generating  $\text{dist}\#2$  and  $\text{result}\#2$  in region C and replacing  $\text{matrix2}\#1$ , in the second position, by  $\text{disti}\#3$ .

In addition to this overhead in PROV and PROV-Dictionary, we use two extra *wasDerivedFrom* edges for every new collection entity to indicate that they derive both from the collection before the update and from the inserted value ( $ikj\#2$ ). Thus, in PROV, this operation has an overhead of 3 *entities*, 6 *wasDerivedFrom*, and 9 *hadMember*, and in PROV-Dictionary, this operation has an overhead of 3 *entities*, 6 *wasDerivedFrom*, and 3 *derivedByInsertionFrom*. Moreover, these overheads depend on the number of elements in the collections and the number of references to them.

Versioned-PROV does not suffer from these problems. It uses checkpoints to indicate multiple versions of a collection, and *derivations by reference* to indicate that two or more entities represent the same collection. In region C of Figure A.4,  $\text{matrix2}\#1$  was defined at checkpoint 2 with the entities 2, m, 0 as members. This changed at checkpoint 36 since this part assignment put  $\text{disti}\@j\#3$  in the first position. Thus,  $\text{matrix2}\#1$  has a version with the members 2, m, 0 between checkpoints 2 and 35, and a version with the members 2,  $\text{disti}\@j\#3$ , 0 after checkpoint 36. Note that in Figure A.4 we show the first value representation of collections for easy reading, but other Versioned-PROV implementations are free to decide on having the *value* attribute or not.

The aforementioned versions are valid for all the entities that derive by reference from  $\text{matrix2}\#1$ . In Figure A.4,  $\text{dist}\@i\#2$  derived by reference from  $\text{matrix2}\#1$ , and  $\text{disti}\#2$

derived by reference from `disti#2`. By transitivity, `disti#2` derived by reference from `matrix2#1`. This derivation avoids the creation of `disti#3` and all the other entities and relationships that exist in the other mappings.

Since an entity can represent multiple versions of a collection in Versioned-PROV, we also use the *checkpoint* attribute in the use of `disti#2` to indicate the used version. Note in region A of Figure A.4 that this operation is using `disti#2` at checkpoint 35 to generate `disti@j#3` at checkpoint 36.

Every entity can only be derived by a single reference: if the algorithm assigns a new value to the variable `disti` (in line 12 of Figure A.1), we must create a new entity (e.g., `disti#3`) as a placeholder for the new value. That is, the *checkpoint* attribute does not apply for reusing an entity with different values. A variable entity in Versioned-PROV represents not just the variable name, but a pair consisting of the variable name and its value (memory area). Note that we do not need a new entity for `disti#2` in the part assignment as it still references the same memory area after the operation.

Finally, `disti@j#3` derived by reference from `ikj#2` in region A of Figure A.4. Since these entities are not collections, the derivation by reference has no impact on them – we use it just for consistency among all derivations. However, this specific derivation has other attributes in addition to *type* and *checkpoint*. We also indicate that it is a write *access* that puts the derived entity in the *key* position 1 of the collection `disti#2`. This information is required to answer the provenance query of *Floyd-Warshall* without encoding matrix positions into entities. Note that the members of `matrix2#1` in region B of Figure A.4 are the actual entities that exist in line 5 of Figure A.1, while the members of `matrix2#1` in Figure A.2 and Figure A.3 are dummy entities that encode the matrix position.

## A.4 Evaluation

We evaluate the space overhead of Versioned-PROV in comparison to plain PROV and PROV-Dictionary by measuring the number of PROV-N statements each approach requires in similar situations. We analyze both the running example and the general case.

**Space overhead analysis of the running example.** For most operations, the storage requirements are the same in all three approaches. The only differences were observed in data structures definitions (lines 2-5 of Figure A.1), reference assignments or accesses (lines 2-5, 7, 9, 12, 18), and data structure updates (line 17).

In (PIMENTEL et al., 2018a) we present the complete provenance graph of *Floyd-Warshall* in these three mappings, coloring only nodes and edges related to the list definitions, reference derivations, and part assignments, since these differ in the mappings. All nodes and edges that are common to all mappings are in light gray. PROV has many colored edges all over the graph due to the aforementioned problems P1 and P2. PROV-Dictionary has fewer scattered edges in the graph, but it has a huge concentration of Dictionary entities that derive from a single *EmptyDictionary* entity due to problem P2. Finally, Versioned-PROV has fewer colored nodes and edges since it does not suffer from these issues.

In Figure A.5(A) we count how many nodes are specific to each approach. Note that PROV and PROV-Dictionary use respectively 7.52 and 4.14 times the number of specific PROV-N statements used by Versioned-PROV to represent the same data structures. Additionally, Versioned-PROV does not impose any node overhead. All of its overhead occurs in edges that specify the membership of collections. On the other hand, PROV and PROV-Dictionary impose node overhead to indicate the position of elements in data structures and to derive immutable entities from existing ones. Moreover, by comparing Figure A.5(A) with Figure A.5(B), which shows the total number of statements, we can see that 29% of PROV statements, 18% of PROV-Dictionary statements, and 5% of Versioned-PROV statements are the overhead caused by collection operations.

These results refer to a small *Floyd-Warshall* execution, with a  $3 \times 3$  matrix representing the input graph. Since the overheads of PROV and PROV-Dictionary grow in terms of the number of collection elements and the number of shared references, more complex input graphs and algorithms can cause a much larger overhead.

**Space overhead analysis of the general case.** In Section A.3, we describe the part assignment of PROV, PROV-Dictionary, and Versioned-PROV. Figure A.6 presents the growth of statements in the three approaches for part assignments. Versioned-PROV has an overhead of **2 PROV-N statements**: the *hadMember* that puts the member in the collection, and the *used* that indicates the changed collection. Plain PROV has an overhead of  $(3 + N) \times R$  **statements** for collections with  $N$  members and  $R$  references: it creates  $R$  *entities*, each of them with 2 *wasDerivedFrom* and  $N$  *hadMember*. Finally, PROV-Dictionary has an overhead of  $4 \times R$  **statements**: it creates  $R$  *entities*, each with 2 *wasDerivedFrom* and 1 *derivedByInsertionFrom*. Note that both plain PROV and PROV-Dictionary also use the changed collection, but this *used* relationship can be inferred from one of the additional *wasDerivedFrom* statements. Hence, we count it only as an overhead for Versioned-PROV. The number of statements for PROV and PROV-Dictionary are lower bounds. If we update a collection  $x$  that is also a member of another

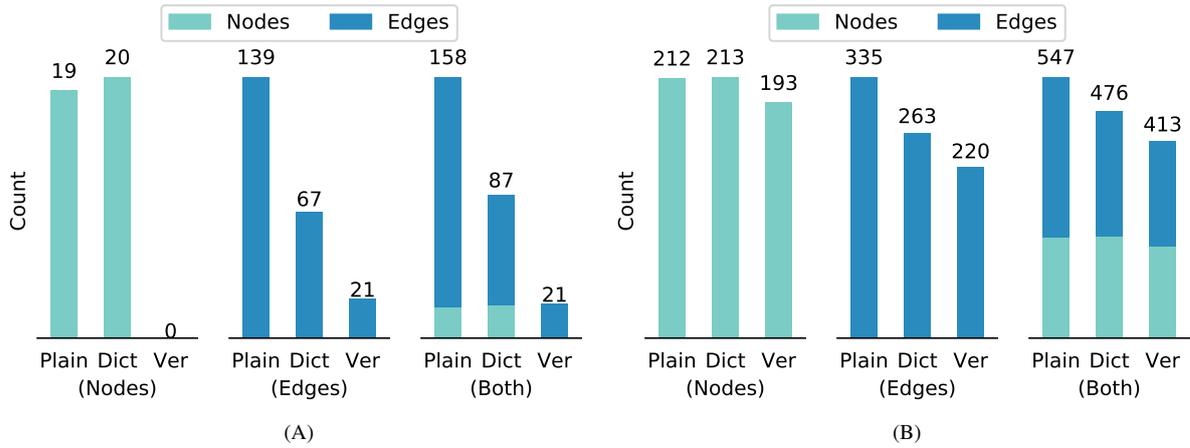


Figure A.5: Number of PROV, PROV-Dictionary, and Versioned-PROV PROV-N statements for list definitions, reference derivations, and part assignments (A) and total number of statements (B).

collection  $\gamma$ , we must also update all the references of  $\gamma$  and apply this same rule with respect to references and number of elements. This occurs in our example of Section A.3.2: the update of `dist#2` with  $R = 1$  and  $N = 3$  motivates the update of `dist#1` with  $R = 2$  and  $N = 3$ .

Besides part assignments, the approaches also differ in list definitions and derivations by reference. Figure A.7(A) shows the overhead of defining a list in each approach. Versioned-PROV has an overhead of only  $N$  *hadMember* statements to define a list with  $N$  elements since they indicate the members with their positions in the list and we reference these positions in accesses. Thus, the provenance of *Floyd-Warshall* in Versioned-PROV includes the accessed positions, allowing us to use these positions to reconstruct the paths of the graph.

On the other hand, plain PROV and PROV-Dictionary have overheads of  $3 \times N + 2$  statements, and  $1$  (global)  $+ 2 \times N + 3$  statements, respectively. This occurs because these approaches do not indicate the access position and the access derivation directly from the member. Hence, we must encode the position information into entities. This encoding requires the creation of  $N$  dummy entities. Each one of these dummy entities derives from their respective entities (i.e.,  $N$  *wasDerivedFrom*) by the application of a new *definelist* activity. The resulting list entity is also generated by this activity (i.e.,  $1$  *wasGeneratedBy* and  $1$  list entity itself), and it has the dummy entities as members. PROV-Dictionary expresses the membership with a single *derivedByInsertionFrom* statement from a single global *EmptyDictionary*, while PROV additionally requires  $N$  *hadMember* statements to define the membership of all elements.

Figure A.7(B) compares the growth of overhead in derivations by reference. Versioned-PROV imposes **no statement overhead** since it uses attributes of *wasDerivedFrom* to indicate

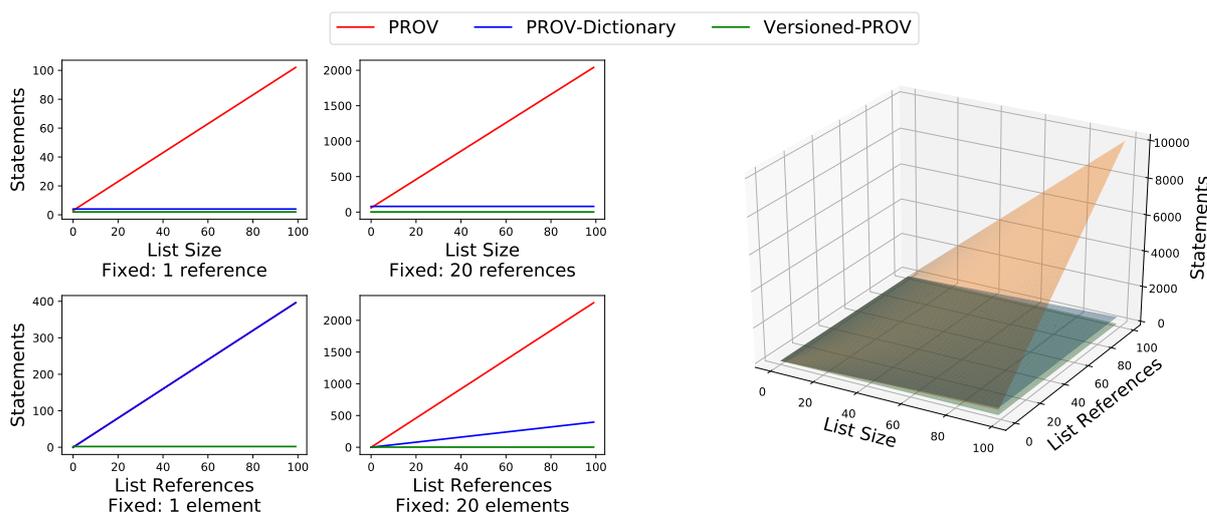


Figure A.6: Overhead functions of part assignments.

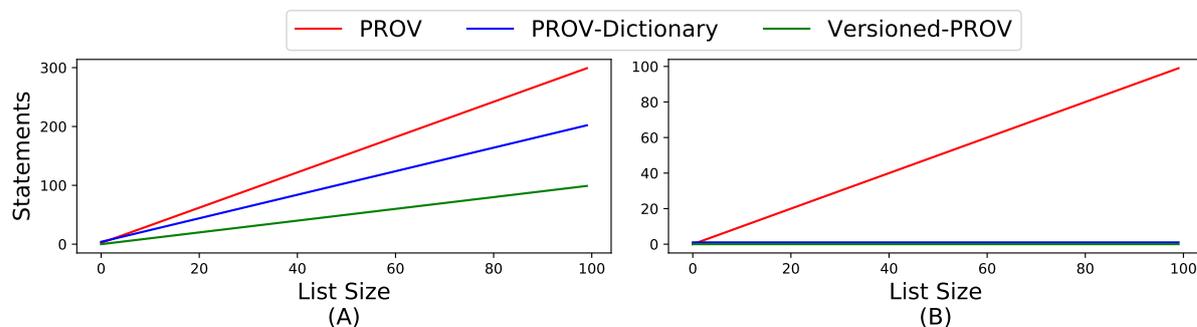


Figure A.7: Overhead functions for list definitions (A) and derivations by reference (B).

the derivation. On the other hand, PROV and PROV-Dictionary have to recreate the membership of this new entity. PROV requires  $N$  *hadMember* statements, and PROV-Dictionary requires a *single derivedByInsertionFrom* statement. Note that both PROV-Dictionary and Versioned-PROV do not grow in terms of the number of elements, but Versioned-PROV still performs better than PROV-Dictionary, since the former does not require any extra statement.

## A.5 Final Remarks

In this appendix, we describe Versioned-PROV, a PROV extension that supports mutable data structures. Tools that collect fine-grained provenance from scripts can use Versioned-PROV to support the collection of provenance from complex data structures and variables that are implicitly modified due to the existence of other variables pointing to the same mutable data. In fact, we use it for noWorkflow 2 (see Chapter 5). Nevertheless, our extension is not restricted to scripts.

The Versioned-PROV approach has some limitations. First, while our extension reduces

the storage overhead for provenance collection from scripts, it introduces an extra overhead for querying due to the requirement of unfolding data structure versions based on checkpoints. Thus, users must consider this tradeoff according to their needs. Second, by using a dictionary-like structure to represent lists (i.e., indexes mapped to keys, and elements mapped to values), some operations still produce an overhead in the provenance storage. For instance, inserting an element at the beginning of a list will require updating all the other members of the list. Third, using an explicit checkpoint ordering imposes synchronization challenges for parallel provenance collection. Finally, the usage of optional attributes to extend PROV imposes a storage overhead in disk due to the attribute name repetition. However, this overhead may not occur depending on how it is stored. A normalized storage schema would remove the repetitions.

Future work is needed to develop an efficient querying algorithm for Versioned-PROV. We foresee the elaboration of unfolding algorithms that converts Versioned-PROV into plain PROV to improve its interoperability and optimize analyses that require many queries. These algorithms could also run by demand, populating caches of unfolded data structures. Additionally, Versioned-PROV could be improved to improve the incremental membership definition of lists.

Finally, our companion website (PIMENTEL et al., 2018a) contains all the source code used to generate images of this appendix in addition to detailed descriptions of the mapping we applied in each approach, as well as a preliminary query implementation. noWorkflow 2 also has implementations for querying the Versioned-PROV model with some adjustments to consider different types of elements.

## APPENDIX B - Version Model Evaluation

This appendix presents the evaluation of the version model proposed in Chapter 5 for tracking the evolution of trials with their intentions (PIMENTEL et al., 2016b). This version model was proposed in collaborations with Juliana Freire (NYU), Vanessa Braganholo (UFF), and Leonardo Murta (UFF).

We present how noWorkflow answers a set of questions related to experiment evolution analysis for evaluating our version model. We obtained and adapted these from the first Provenance Challenge<sup>1</sup> and ProvBench workshops<sup>2</sup>. We answer the questions using the example described in Section 5.3.4. This example is, in fact, the workflow of the first Provenance Challenge implemented in Python with procedures implemented as “dummies”. The full history of this experiment can be obtained on noWorkflow by running `now demo 2016_ipaw_paper`.

**Q1<sup>1</sup>: if a scientist has executed an experiment twice but has replaced some procedures in the second trial, what are the trial differences? Q2<sup>3</sup>: comparing multiple executions according to their parameters, what are the differences in execution behavior?** noWorkflow supports the comparison of activation graphs from two trials (see Section 5.4.3. Figure 5.16 presents a comparison of activation graphs from trials 1.1.1 and 2.1.1. It is possible to see that “convert” was replaced by “pgmtoppm” and “pnmtjpeg”. To compare execution behaviors according to parameters, we can compare trials that share the same code base but have different parameters.

**Q3<sup>4</sup>: how differences in the input data relate to differences in the output values?** We can use the `now diff -f` command to compare file accesses of trials (as shown in Figure 5.15).

<sup>1</sup><http://twiki.ipaw.info/bin/view/Challenge/FirstProvenanceChallenge>

<sup>2</sup><https://sites.google.com/site/provbench/home/provbench-provenance-week-2014>

<sup>3</sup><https://github.com/provbench/Swift-PROV>

<sup>4</sup><https://github.com/provbench/CSIRO-PROV>

```
1  sqlite> SELECT a.trial_id, a.name, count(a.name) as c
2      FROM activation a
3      INNER JOIN evaluation e ON (
4          a.trial_id = e.trial_id
5          AND a.id= e.id
6      )
7      WHERE repr = "-1"
8      GROUP BY a.trial_id, name
9      ORDER BY c DESC;
```

Figure B.1: Activations that produce a failing value frequently.

This command compares input data, output data, and arguments. Thus, it is possible to get the differences on inputs and compare them to output values by restoring them.

**Q4<sup>4</sup>: using historical provenance, which parts of the execution fail frequently?** If we specify that the return value “-1” of an activation represents a failure, the query presented in Figure B.1 will return the most frequent failures on all trials combined. A more complex query could use the Versioned-PROV model to look at the types (classes) of return values and check for exceptions.

**Q5<sup>5</sup>: which trials are related to a given trial? Q6<sup>5</sup>: a given trial was derived from which trial? Q7<sup>6</sup>: what are the available trials, and what are their durations? Q8<sup>6</sup>: how many trials are associated with a given source code? Q9<sup>6</sup>: how many trials present failures?** Looking at the Evolution History (as shown in Figure 5.7), it is possible to see both the ancestor of a given trial and all trials derived from it. The evolution history also presents all available graphs. To get their duration, a user can activate tooltips on `now vis` or Jupyter Notebook and access trial information, including its duration. To get all trials associated with a given source code, we can filter the history to a specific script. Finally, the history graph presents trials with failures as red nodes.

---

<sup>5</sup><https://github.com/provbench/VisTrails-PROV>

<sup>6</sup><https://github.com/provbench/Wf4Ever-PROV>

## ANNEX A - Git Integration Evaluation

### A.1 Introduction

This annex presents the evaluation of using Git as a content database for noWorkflow. This evaluation was performed by Vynicius Pontes (PONTES, 2018) in his undergraduate capstone project under my co-advisory and Leonardo Murta (UFF) advisory.

As we discuss in Chapter 5, noWorkflow 1 and 2 added support for a content database that uses Git for storing files. This content database has the advantages of compressing files using zlib (GAILLY; ADLER, 2017) and combining objects into packfiles that contain one version of them and deltas from one version to another (CHACON; STRAUB, 2014). For this integration, noWorkflow uses two alternative libraries: *Dulwich* (VERNOOJJ, 2018) (pure Python implementation) or *PyGit2* (IBÁÑEZ et al., 2018) (bindings for a C library) to perform Git operations.

While both libraries produce a Git-compatible content database, they reduce the content database at different rates, and they impose distinct performance overheads. Hence the goal of this annex is to evaluate these aspects of the Git integration.

#### A.1.1 Materials and Methods

In this section, we discuss the methodology we used to collect, prepare, and analyze the overhead of the integration on scientific experiments.

##### A.1.1.1 Research Questions

For analyzing the integration, we defined two research questions:

**RQ.G1.** *Is there any reduction in the size of the content database?* Since the main goal of the integration is to benefit from the compression and packfiles provided by Git, it is important

to evaluate if these characteristics reduce the size of the content database. For answering this question, we measure the size of the content database of each intervention (i.e., Dulwich and PyGit2) and compare it to the size of the baseline (i.e., old content database).

In addition to this comparison, we also evaluate the effect of using Git garbage collection on the reduction of the content database size for projects with files that grow incrementally for each trial.

**RQ.G2.** *Is there any performance overhead with the integration?* For compressing files, calculating their deltas, and organizing them into packfiles, Git necessarily imposes a performance overhead during the execution of the scripts. If the performance overhead is too big, it could make the Git approach unfeasible for experiments. For answering this question, we measure the duration of trials of each intervention (i.e., Dulwich and PyGit2) and compare it to the duration of baseline trials (i.e., old content database).

#### A.1.1.2 Corpus

The main corpus of this experiment is composed of seven scripts. For obtaining these scripts, we first searched for “Python Github” on Google Scholar and obtained three repositories from two experiments (GILL, 2015; WICKERT, 2016) and one library (MCFEE et al., 2015). Then, we extracted three scripts from the experiments and two from the library. Finally, we searched for Python scripts in scientific blog posts and obtained two scripts (COOK, n.d.; WINCKEL, 2014).

To answer the research questions, we run each script four times for each of the three approaches using noWorkflow 1: old content database, git integration using Dulwich, and git integration using PyGit2. Then, we calculate the average of the desired metric (space or time) for each script and use it to compare the approaches. We use the old content database as a baseline for the comparisons.

In addition to these scripts, we also built a synthetic experiment script that simulates a trial-and-error process by appending random text content (around 13 Megabytes) to a text file in every trial. This script aims to simulate multiple trials of an experiment that grows the size of a file over the trials. We execute this script ten times, increasing the size of a specific file in each trial. Then, after ten trials, we execute the `now gc` command that uses Git for applying the garbage collection over the content databases and creating packfiles with deltas.

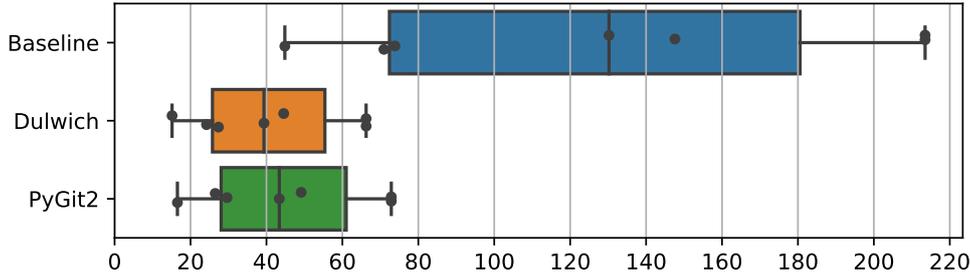


Figure A.1: Size of content database directories of all script executions for each content database type [adapted from Pontes (2018)].

## A.1.2 Results

### A.1.2.1 RQ.G1. Is there any reduction in the size of the content database?

**Real scripts.** As stated before, we executed four times each one of the seven scripts and calculated the average of the content database size to answer this research question. Table A.1 presents the results of this part of the experiment by script, indicating the average reductions compared to the baseline and Figure A.1 condenses the results into box plots for comparison. Note that the Git content databases greatly reduced the storage, producing content databases that are 59.9% to 69.8% smaller than the baseline (average of 65.23% for PyGit2 and 68.35% for Dulwich).

Table A.1: Average sizes of the content database after 4 executions for each content database type and reduction percentage between the Git content database and the baseline [adapted from Pontes (2018)].

Script	Avg. Size (MB)			Avg. Reduction (%)	
	Dulwich	PyGit2	Baseline	Dulwich	Pygit2
<code>analyse.py</code> (GILL, 2015)	27.348	29.620	73.840	-62.96%	-59.89%
<code>beat_tracker.py</code> (MCFEE et al., 2015)	66.256	72.888	213.496	-68.97%	-65.86%
<code>estimate_tuning.py</code> (MCFEE et al., 2015)	66.252	72.884	213.492	-68.97%	-65.86%
<code>gflex.py</code> (WICKERT, 2016)	44.544	49.144	147.596	-69.82%	-66.70%
<code>menger_sponge.py</code> (COOK, n.d.)	24.212	26.508	70.944	-65.87%	-62.64%
<code>qho2.py</code> (WINCKEL, 2014)	39.344	43.380	130.230	-69.79%	-66.69%
<code>source.py</code> (GILL, 2015)	15.142	16.547	44.844	-66.23%	-63.10%
<b>Total</b>	283.098	310.971	894.442	<b>-68.35%</b>	<b>-65.23%</b>

Next, we used a hypothesis test to verify these results statistically. With this goal, we first checked the normality of these results using the Shapiro Wilk Test (SHAPIRO; WILK, 1965). We found that all distributions follow a normal distribution since they failed to reject the null

hypothesis at 95% confidence (p-values for baseline, Dulwich, and PyGit2 were 0.307, 0.405, 0.398, respectively).

Since the distributions are normal, we compared both interventions to the baseline using the Paired T-Test (STUDENT, 1908), with the null hypothesis that the averages are equal. We first compared Dulwich to the baseline and found a p-value of 0.003, rejecting the null hypothesis at 95% confidence. Then, we compared the PyGit2 to the baseline and found a p-value of 0.031, rejecting the null hypothesis at the same confidence again. Thus, we can confirm that the Git integration reduces the size of the content database.

For estimating how large the difference is, we calculated the effect sizes using Cohen’s  $d$  (HEDGES; OLKIN, 2014), and we obtained  $d$  estimates of 1.73129 and 1.63806 for Dulwich and PyGit2, respectively. These effect sizes are considered large, indicating a great reduction in the content database size.

**Git Garbage Collection.** In the second part of the experiment, we evaluated the growth of the content database and the effect of using garbage collection on the content database. We executed a synthetic script ten times for each content database, resulting in ten consecutive trials that increased the size of a file sequentially. Then, after the executions, we ran `now gc` to execute the Git garbage collection on the content database. Table A.2 presents the size evolution of the content database over the 10 trials and after the execution of the `now gc` command. Note that the sizes of the content databases that use Git are always smaller than the baseline. Additionally, the `now gc` operation reduces the size of the Git content databases by 73.79%, while the baseline does not support this feature.

Table A.2: Size in Megabytes of content database directory for each noWorkflow trial using the synthetic script and the execution of `now gc` at the end [adapted from Pontes (2018)].

	T.1	T.2	T.3	T.4	T.5	T.6	T.7	T.8	T.9	T.10	GC
<b>Baseline</b>	18.1	44.2	83.3	135.3	200.5	278.6	369.7	473.9	591.1	721.3	721.3
<b>Dulwich</b>	12.2	32	61.6	101.1	150.4	209.6	278.7	357.6	446.4	545.1	189.1
<b>PyGit2</b>	12.6	32.8	63	103.4	153.8	214.3	284.9	365.6	456.3	557.1	189.1

The reason for this high reduction may be associated with the nature of the text file generated by our synthetic script. Text files may achieve a good compression and efficient deltas on Git, while other formats may suffer from the lack of compression. We did not execute `now gc` for the real scripts on the first part of this experiment because they only contain a single version of the scripts. The garbage collection usually benefits the most from multiple versions of the same file.

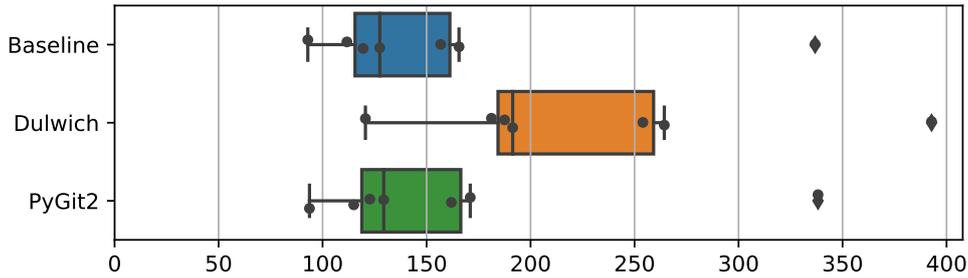


Figure A.2: Duration of all script executions for each content database type version [adapted from Pontes (2018)].

### A.1.2.2 RQ.G2. Is there any performance overhead with the integration?

For this research question, we only considered the experiment with real scripts. In this case, we calculated the average of the duration of four trials for each one of the seven scripts and each content database. Table A.3 presents the results of this part of the experiment by script, indicating the average performance overhead compared to the baseline, and Figure A.2 condenses the results into box plots for comparison.

Note that PyGit2 is much faster than Dulwich, but has almost the same effect in reducing the storage size when compared to the baseline. Additionally, it only imposes 0.4% up to 3.3% (average of 1.9%) performance overhead. On the other hand, Dulwich imposes a performance overhead from 16.6% up to 62.31% (average of 43.32%). The reason for this discrepancy is probably related to the fact that Dulwich is a pure Python implementation, while PyGit2 is a C implementation with Python bindings.

We also used hypothesis tests to verify these results statistically. Thus, once again, we checked the normality of the results using the Shapiro Wilk Test (SHAPIRO; WILK, 1965).

Table A.3: Average execution duration after 4 executions for each content database and average differences between Git content database and baseline [adapted from Pontes (2018)].

Script	Avg. Duration (Sec)			Avg. Difference (%)	
	Dulwich	PyGit2	Baseline	Dulwich	PyGit2
<b>analyse.py (GILL, 2015)</b>	181.2	115	111.7	62.31%	3.02%
<b>beat_tracker.py (MCFEE et al., 2015)</b>	254	161.9	156.8	61.94%	3.22%
<b>estimate_tuning.py (MCFEE et al., 2015)</b>	264.3	171	165.6	59.64%	3.26%
<b>gflex.py (WICKERT, 2016)</b>	187.5	122.7	119.5	56.97%	2.69%
<b>menger_sponge.py (COOK, n.d.)</b>	392.8	338.2	336.8	16.63%	0.40%
<b>qho2.py (WINCKEL, 2014)</b>	191.4	129.4	127.5	50.10%	1.43%
<b>source.py (GILL, 2015)</b>	120.6	93.7	92.9	29.86%	0.94%
<b>Total</b>	1591.8	1131.9	1110.8	43.32%	1.90%

However, in this case, we rejected the null hypothesis for all duration distributions at 95% confidence (p-values for baseline, Dulwich, and PyGit2 were 0.010, 0.014, and 0.014, respectively).

Since the distributions are not normal, we compared both interventions to the baseline using the Wilcoxon Signed-rank Test (WILCOXON, 1992), a non-parametric test. Once again, our null hypothesis is that the averages are equal. When we compared each intervention to the baseline, we found the same p-value: 0.016. Hence, we reject both null hypotheses at 95% confidence, indicating that both PyGit2 and Dulwich impose some performance overhead on the baseline.

For estimating how large the differences are, we calculated the effect sizes using Cliff's Delta (CLIFF, 1996). When comparing the effect size of Dulwich to the baseline, we obtained a delta of -0.63265, which is considered large. However, for the effect size of PyGit2, we obtained a delta of -0.14285, which is negligible.

## A.2 Conclusion

In the experiments, we observed that PyGit2 could greatly reduce the content database size (65.23% on average), with a negligible impact on the performance (1.90% on average). We also observed that garbage collection could further reduce the content database size (73.79% in the synthetic script experiment).

We also observed that despite reducing the content database size as much as PyGit2, Dulwich imposes a large performance overhead (43.32% on average). Hence, when both libraries are available, it is preferable to use PyGit2 for the integration in noWorkflow 1 and 2.

However, since PyGit2 only provides Python bindings for a C library, its installation is harder than the installation of Dulwich, as it is a pure Python implementation of Git operations. Hence, it is worth it to support both libraries in noWorkflow 1 and 2 for broader compatibility.

Despite the positive results in the experiments, it is worth mentioning that these experiments are not conclusive for many reasons. First, we used a small set of experiments that are not representative of all the experiments that might produce files for content databases. Second, we evaluated it with small files in the magnitude of Kilobytes and Megabytes, which are reasonable for Git, since it was designed for source code. The compression and packing algorithms may struggle to operate bigger files. Finally, the experiments were performed in 2018, and all the tools involved in the experiment (i.e., Git, Dulwich, PyGit2, and noWorkflow) evolved since the performance overheads and size reductions were measured.