Reducing the Storage Overhead of the noWorkflow Content Database by using Git

Vynicius Morais Pontes*
João Felipe Pimentel**
Leonardo Murta***
Troy Kohwalter****
Daniel de Oliveira*****

Abstract

noWorkflow is an open-source tool that non-intrusively captures provenance data from Python scripts, allowing researchers to analyze and manage it in different ways. noWorkflow captures accessed files and modules and stores them in the file system. It uses the provenance to form a version timeline that lets the researcher assess the evolution of his trials. This approach introduces size overheads by the way that noWorkflow handles the file storage through each version. Any tiny change to a file requires storing the new file as a whole. We propose an alternative to this file system storage by introducing Git tools to improve the way that noWorkflow stores the raw data, reducing the storage size overhead by an average of 65.23%, with the tradeoff of adding a small processing time overhead of 1.90% on average.

Keywords: experiments, provenance, scripts, noWorkflow, Git.

1 INTRODUCTION

Among scientists, script languages such as Python are a popular and accessible tool for automation of *in-silico* experiments. Scripts provide a systematic view of the execution path, allowing an easy adaptation of the data

* Information Systems undergraduate student at Universidade Federal Fluminense; [1] Universidade Federal Fluminense, Av. Gal. Milton Tavares de Souza, s/nº, Boa Viagem, *Campus* da Praia Vermelha, 24210-346 – Niterói, RJ; vyniciusmorais@id.uff.br
** João Felipe Nicolaci Pimentel; Advisor; PhD student at Universidade Federal Fluminense; [1]; jpimentel@ic.uff.br
*** Leonardo Gresta Paulino Murta; Advisor; Associate Professor at Universidade Federal Fluminense; [1]; leomurta@ic.uff.br
**** Troy Costa Kohwalter; Examination board; Postdoc at Universidade Federal Fluminense; [1]; tkohwalter@ic.uff.br
***** Daniel Cardoso Moraes de Oliveira; Examination board; Professor at Universidade Federal Fluminense; [1]; danielcmo@ic.uff.br

flow. Provenance provides an important support to the development process of modeling scientific experiments using scripts, giving scientists useful information about the execution steps and inputs to compose the generated data (DEY et al., 2015). In that scenario, noWorkflow comes as a tool that transparently and non-intrusively captures and stores provenance from Python Scripts.

For each experiment trial (*i.e.*, an execution of the experiment), noWorkflow collects imported modules, environment variables, function calls, file accesses, variables and associates this provenance data to a sequential trial identification number (MURTA *et al.*, 2015; PIMENTEL *et al.*, 2017). The trial identification number makes every execution unique, associating the experiment files versions to the trial. To store the provenance, noWorkflow uses a relational database and a file-based storage called “content database”. Furthermore, noWorkflow provides tools for iterating through the trials, analyzing the evolution of the files and comparing the diffs.

This approach of storing several raw files in the content database brings some size overheads because noWorkflow does not apply any special treatment to the file, like compressing them or organizing in chunks to avoid storing the same data multiple times. Fortunately, the way that noWorkflow works to store files in the file system and relate them to a version identifier is very similar to a Version Control System (VCS) like Git. noWorkflow, as Git, calculates the content hash of the file, stores the content inside the repository and index it using the generated hash. Thus, if there are two versions with the same content, there will be no increase in the disk space required for its storage (CHACON; STRAUB, 2014). When Git commits modifications to the file system and associates them to an identifier, it generates a lineage of those files, where it is possible to analyze and navigate through them, recovering old versions if necessary. Moreover, besides offering multiple advantages like working with multiple developers and parallelize the development through branches, Git can work as a powerful file database providing tools for compacting and organizing files in chunks, enabling the reuse of disk space and, consequently, reducing the occupied storage space.

Unlike Git, noWorkflow does not compress files and does not compute deltas. Thus, tiny changes on files require noWorkflow to store the new versions all over again. With this idea in mind, we propose a way to integrate Git into noWorkflow. We opted to use Python libraries that implement Git natively, so we

could use low-level tools of it (called plumbing API in Git parlance). The results were satisfactory in reducing considerably the storage size overhead by 65.23% on average, with a tiny processing time overhead of 1.90% on average, due to Git tools setup and its compaction time.

This article is organized as follows. Section 2 describes how the noWorkflow file system interface works. Section 3 presents how we replace the noWorkflow content database by Git. Section 4 explains the experiments planning and results for assessing the benefits of using Git. Section 5 discusses the related work. Finally, Section 6 concludes the article.

2 NOWORKFLOW PROVENANCE STORAGE

noWorkflow provides a solution for the job of collecting and organizing information about execution steps, the execution environment, data inputs, and outputs of Python scripts. noWorkflow uses several techniques like reflection, tracing, profiling, among others to collect provenance. Furthermore, with the provenance data persisted, noWorkflow gives users a set of tools to analyze and query the collected provenance, such as SQL and Prolog queries, and graph visualizations (PIMENTEL *et al.*, 2017). Hence, noWorkflow is built based on three key components *Provenance Collection, Provenance Storage, and Provenance Analysis* as presented in Figure 1. In this article, we focus only on the *Provenance Storage* module.

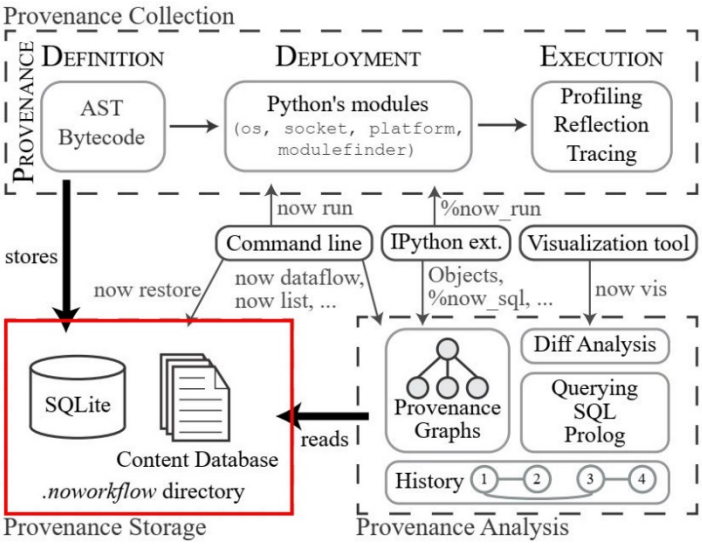

Figure 1 – noWorkflow Architecture. Obtained from Pimentel et al. (2017).

The *Provenance Storage* module is a component in the noWorkflow architecture that deals with information storage. It allows other components to store and access provenance data. This component consists of an SQLite relational database and a file-based storage called *content database*. During each trial execution, the *content database* stores accessed files, modules, and scripts in the file system. For storing a file, the content database calculates its SHA1 hash (DANG, 2015) and copies its content to a new file identified by the hash. After that, it stores the hash in the relational database, so the content can be referenced when needed. The relational database also keeps the trial identification, variables, function names, function calls, and modules access (PIMENTEL *et al.*, 2017).

The *content database* works as a file system interface. It is responsible for storing and recovering all file-related data (MURTA *et al.*, 2015; PIMENTEL *et al.*, 2017). Storing versions of accessed files is important for several reasons. Besides the script files itself, experiments often use files as inputs as well as outputs, such as images, spreadsheets, text files, binaries, among others. Thus, keeping these versions is useful for diff analyses and reproducibility by restoring old versions of the script, modules, input, and output files.

The *content database* interface has two main functions: *put* and *get*. The *put* function receives a file content as a parameter, computes the SHA1 hash (DANG, 2015) of it, and stores the content using the hash as the name of the directory (first two digits of the hash) and file (remaining digits of the hash). Consequently, files with the same content (which would generate the same hash), are not stored more than once. The *get* function receives as a parameter a SHA1 hash (DANG, 2015), so the *content database* recovers and return the related file content.

However, any tiny change on a file would imply changes on the hash, demanding to store the new file as a whole. Consequently, this approach does not give a proper treatment for big files or files that grow along with several trials. For example, suppose an experiment that reads a spreadsheet and appends thousands of lines to it in each trial. After multiple trials, noWorkflow will store every file version, even though the last version contains the content of all previous versions. In this scenario, the occupied space could be a problem when the experiment runs with even bigger files.

3 INTEGRATING GIT

This article has the goal of improving the file storage of noWorkflow by using Git. Git is a popular Version Control System that does not need a central repository. It builds the entire file repository locally, with all the file history. This enables Git to perform almost all of its operations with a local database without an internet connection, and when appropriate, the user can push his local modifications to a remote database server (CHACON; STRAUB, 2014). To fulfill this objective, Git must deal with file storage in an efficient way, because the overhead of storing every file version could cause many problems to large projects.

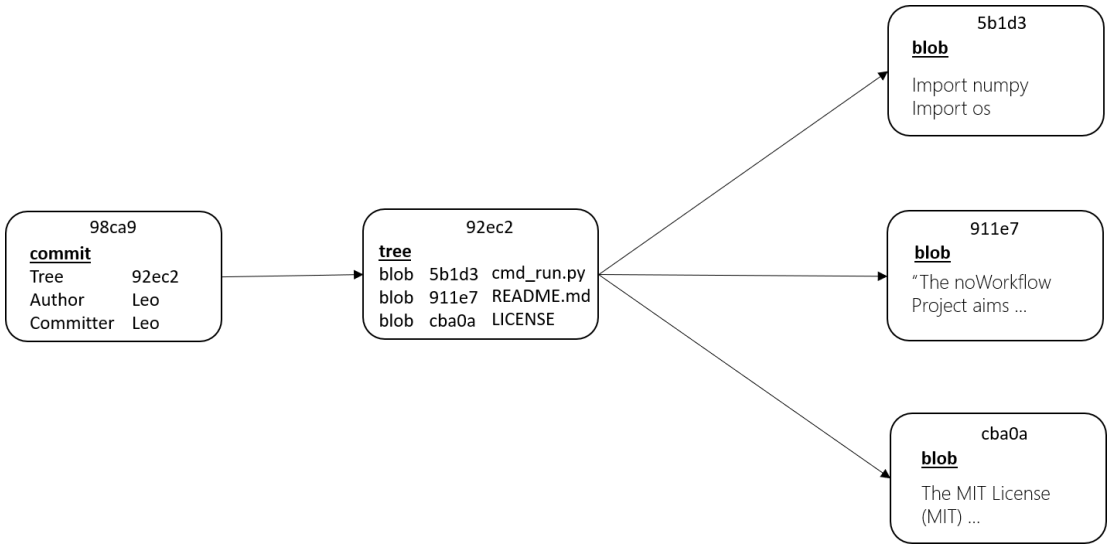

Figure 2 – Git objects. Adapted from CHACON and STRAUB (2014).

Git stores several kinds of objects inside the repository to build its structure. The main ones are: blobs, trees and commit objects. Every object is named with a SHA1 hash (DANG, 2015) of its content, making them unique. Generally, the body of the object contains hashes referencing to other objects inside de repository. The blob object is a binary object that carries the content of a file. The tree object stores a key-value map of hashes to indicate the blob and tree objects inside de repository and link them with the corresponding file name. The commit object contains a hash for a tree object, the name of the author and the commit information about the date, the message of the commit, and optionally hashes to the parent commits. Figure 2 presents a commit object that references

a tree object. The tree object has three references to blob objects through hashes. Git uses the parent hashes in commit objects to build a timeline of commits, as presented in Figure 3.

Figure 3 – Git commits and their parents. Adapted from CHACON and STRAUB (2014).

Git uses two techniques to reduce its size overhead. First, it compresses every new content that enters the repository with zlib (ADLER, 2018). Second, it packs repository objects into packfiles that stores only a single version of similar files and their deltas from one version of the file to another (CHACON; STRAUB, 2014). Consequently, when a repository contains a big file that has suffered many additions over the time, instead of keeping several versions of the big file, Git keeps only the chunks that can build any desired version. This command of packing the files and cleaning the repository is called *garbage collection* (CHACON; STRAUB, 2014).

There are two interfaces to interact with Git: the *Porcelain* and the *Plumbing* (CHACON; STRAUB, 2014). The *Porcelain* is the user-friendly interface used by developers to interact with the Version Control features of Git. The *Porcelain* interface includes commands like *git add, git commit, git push, git checkout* and others. The *Plumbing* interface contains commands that can be classified as low level. Those commands allow us to manually interact with the Git objects we have presented. That means we could create, integrate and remove the Git objects inside the repository without necessarily using Git as a Version Control System. As stated by Chacon and Straub (2014) “The core of Git is a simple key-value data store”, so it is able to be used as a file database.

Since its possible to interact with Git file database features and take advantage of Git techniques to reduce size overhead, we can use Git itself to manage the provenance storage of noWorkflow. We envisioned two approaches

to integrate Git to noWorkflow. The first one is to enable noWorkflow to do system calls to the Git installed on the users' machine. However, as noWorkflow does thousands of calls to *content database* in one single trial, this approach has proven impracticable, due to huge memory and time overhead. The other approach is using libraries that implement Git in native Python. We found and integrated two: *Pygit2* ("pygit2", 2018) and *Dulwich* ("dulwich", 2018).

The integration starts in the transformation of the *content database* directory into a Git bare repository. Bare repository means that there is no workspace area with the files of a specific version uncompressed for edit, that is, Git will not interact with files outside of the repository folder. The libraries abstract all the Git objects described before as Python classes. This simplifies the connection and creation of repositories, commits, trees, and blobs.

The noWorkflow *content database* API has three main functions: *connect*, *put*, *get*. We added a new function called *commit content*. Table 1 shows the key changes to allow Git integration. The *put* function was one of the biggest changes of this integration because the libraries do the compression exactly like Git does before inserting into the repository. This brings an advantage of reducing the size of stored files but comes with a time overhead that we discuss in the next section. To help to reduce the time overhead, we use a process queue to do the IO job. Before, there was no parallel job with the IO operations. Now, when the *put* function is called, we deliver to a process queue the responsibility to call the Git library to compress and store the data. This is done in parallel with the main execution thread of noWorkflow. All the manipulation with *content database* and Git tools is during a trial is automatically done by noWorkflow own instructions without any user interaction.

The *garbage collection* function is a tool that needs to be called manually on the Git repository. None of the used libraries, PyGit2 ("pygit2", 2018) and Dulwich ("dulwich", 2018), support the *garbage collection* function. So, in this case, we need to call the Git installed in the user's machine. We added a command called *now gc* which calls the local Git to execute the garbage collection in the noWorkflow repository. If Git is not installed, the command shows a warning stating that to use this function Git needs to be installed. Notice that all other functions will work without Git installation.

Table 1 – Content database main functions before our work and after, with Git.

Function	Before this work	With Git
Connect	Create the *content database* directory if it does not exist.	Initialize a bare Git repository if it does not exist and create an initial commit without any content.
Put	Receives a file content as a parameter, computes the SHA1 hash of it, and stores the content using the hash as the name of the directory (first two digits of the hash) and file (remaining digits of the hash).	Receives a file content and the file name as a parameter. The function put the file content into a process queue which will asynchronously call the Git library to compress and persist the content to the repository. In parallel, the function compute and returns the SHA1 hash, so the main process continues.
Get	Given a hash, search for a file and return its content from the *content database* directory.	Given a hash, search for a file and return its content from the repository.
Commit Content	Does not have.	After all processes finish their job of putting files into the repository, the *commit content* function creates a tree object containing all hashes and names of added blobs and attaches that tree to a commit object. Then, the function fulfills the commit object with the previous commit object hash, an optional commit message and adds it to the repository.

Table 2 – Setup of content database engine based on environment found.

	Old Content Database Found	Git Content Database Found	No Content Database Found (new experiment)
Only Dulwich Installed	Old	Dulwich	Dulwich
Full installation	Old	PyGit2	PyGit2
Only Old Version Installed	Old	X	X

As we are changing the organization of the stored files, we recognize if the current *content database* directory is in the old format. In this case, we use the old *content database* engine without changing anything on the environment. Table 2 shows how the setup works for each possible situation. Depending on what is installed in the computer that is running noWorkflow (rows) and the format

of the noWorkflow content database (columns), each cell indicates the mechanism that noWorkflow employs for accessing the content database. We choose PyGit2 as our main library due to a smaller processing time overhead than Dulwich, but still with a good storage space reducing capability. We discuss more details in session 4.

4 EVALUATION

To evaluate the integration, we need to answer two research questions: Is there any size reduction of the *content database*? Is there any time overhead with the integration? Our main goal is to decrease the size of the *content database*, so the size of the repository is the main indicator we are monitoring. The second one is the time of the execution. If the time overhead is exorbitant it could make the Git approach unfeasible. We evaluate our approach with real experiments to check if the proposed changes result in improvements (COOK, 2011; GILL, 2015; “librosa”, 2018; WICKERT, 2016; WINCKEL, 2014). We also use a synthetic experiment that appends several text contents to a file to simulates the size growing of an experiment workspace.

Our corpus is composed of eight experiments. We implemented a script for automating the execution of noWorkflow over each experiment. This script executes each experiment four times, collecting the elapsed time and the final size of the repository. The script cleans the noWorkflow directory before each execution. For comparison purposes, we execute the experiments with the old version of noWorkflow, using the old *content database* engine, and the new version, integrated with Git, using the Dulwich (“dulwich”, 2018) and PyGit2 (“pygit2”, 2018) libraries. After collecting the data, we plot the average of the four executions in box plot graphs, so we can analyze the results. We also do some hypothesis tests to statistically verify our proposal.

Storage Overhead. We present the size results in Table 3. Note that the sizes requirements of using Git with Dulwich and PyGit2 are smaller than in the old noWorkflow version, as expected. Figure 4 shows a boxplot graph describing the average size of the four executions for each content database engine. The Git engines reduce considerably the size of the *content database* directory.

Notice that we don't use the *garbage collection* tool in these executions. The reductions happen even without the garbage collection, due to zlib compression (ADLER, 2018) that Git libraries apply.

Table 3 – Average size of the content database after 4 executions for each engine version and reducing the percentage between the Git engines and the old one.

	Avg. Size (MB)			Avg. Reducing (%)	
Script	**Git Dulwich**	**Git PyGit2**	**Old Version**	**Git Dulwich**	**Git Pygit2**
analyse.py (GILL, 2015)	27,348	29,620	73,840	-62.96%	-59.89%
beat_tracker.py ("librosa", 2018)	66,256	72,888	213,496	-68.97%	-65.86%
estimate_tuning.py ("librosa", 2018)	66,252	72,884	213,492	-68.97%	-65.86%
gflex.py (WICKERT, 2016)	44,544	49,144	147,596	-69.82%	-66.70%
menger_sponge.py (COOK, 2011)	24,212	26,508	70,944	-65.87%	-62.64%
qho2.py (WINCKEL, 2014)	39,344	43,380	130,230	-69.79%	-66.69%
source.py (GILL, 2015)	15,142	16,547	44,844	-66.23%	-63.10%
Total	283,098	310,971	894,442	**-68.35%**	**-65.23%**

Table 4 - Average execution time after 4 executions for each engine version and average difference between the Git engines to the old one.

	Avg. Time (Sec)			Avg. Difference (%)	
Script	**Git Dulwich**	**Git PyGit2**	**Old Version**	**Git Dulwich**	**Git Pygit2**
analyse.py (GILL, 2015)	181.2	115.0	111.7	62.31%	3.02%
beat_tracker.py ("librosa", 2018)	254.0	161.9	156.8	61.94%	3.22%
estimate_tuning.py ("librosa", 2018)	264.3	171.0	165.6	59.64%	3.26%
gflex.py (WICKERT, 2016)	187.5	122.7	119.5	56.97%	2.69%
menger_sponge.py (COOK, 2011)	392.8	338.2	336.8	16.63%	0.40%
qho2.py (WINCKEL, 2014)	191.4	129.4	127.5	50.10%	1.43%
source.py (GILL, 2015)	120.6	93.7	92.9	29.86%	0.94%
Total	1591.8	1131.9	1110.8	**43.32%**	**1.90%**

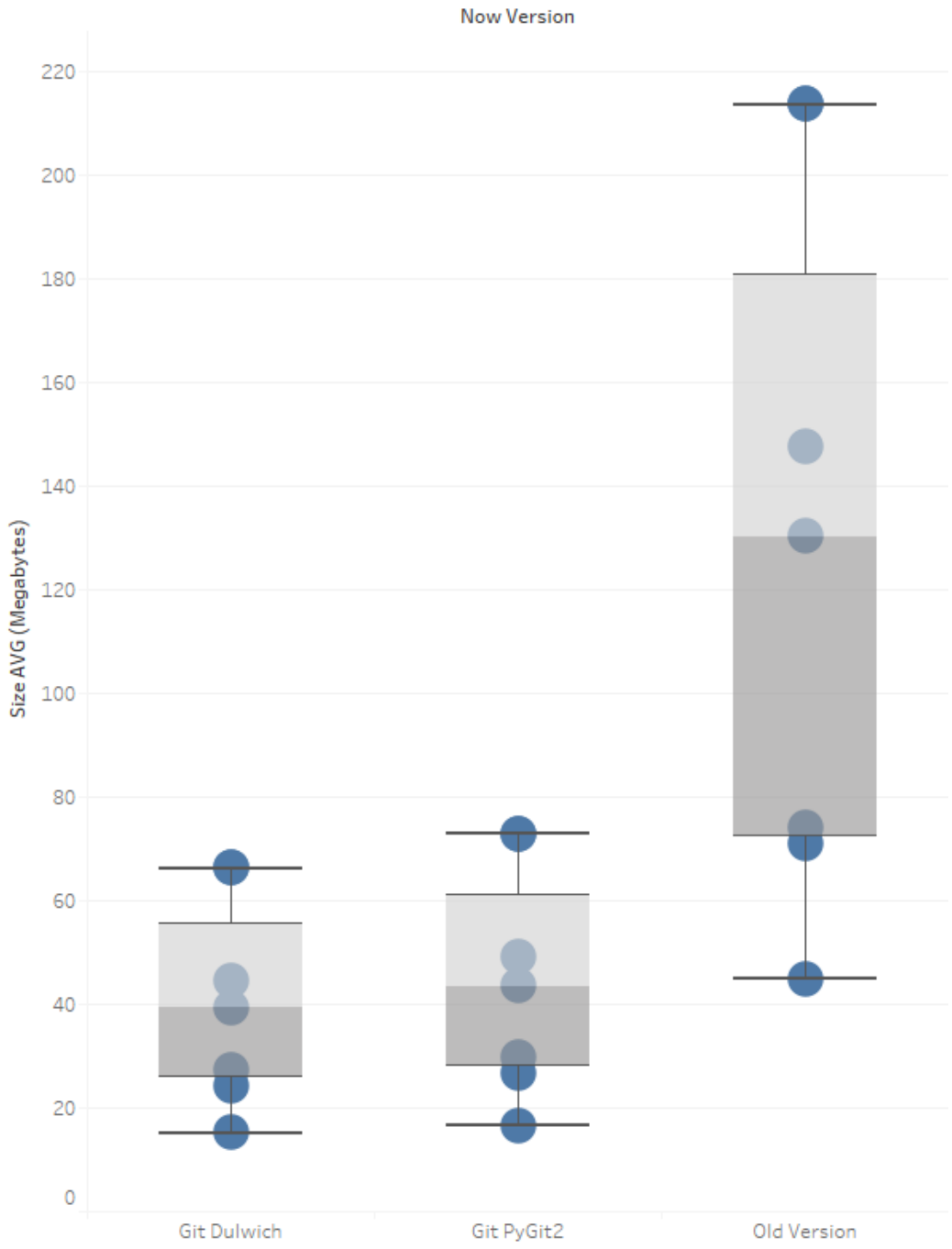

Figure 4 – Size of *content database* directory of all script executions for each *content database* engine version.

We apply to these results a hypothesis test, so we can statistically verify that our proposal improves the storage. To do that we use the Old and PyGit2 versions data shown in Table 3. First, we confirm the normality of these distributions with the Shapiro Wilk test (SHAPIRO; WILK, 1965). As the distributions are normal we use the Paired T-Test ("THE PROBABLE ERROR OF A MEAN", 1908) which is used to compare the means of two sets of paired

samples, taken from two populations with unknown variance. Our null hypothesis is that the averages are equal. The p-value generated is equal to $3.123e-3$, bellow 0.05, meaning that we can reject the null hypothesis and, statistically verify that our changes reduce the size of the content database. We know our changes affect the size of the *content database*. However, to know how much is this effect we apply an effect size test Cohen's d (HEDGES; OLKIN, 1985) for normal distributions. The estimate d is 1.638064 which is considerably large. That means we apply effect on the size of the *content database* and that effect is large.

Time Overhead. Table 4Erro! Fonte de referência não encontrada. shows the processing time results where is possible to see that Dulwich is much slower than the PyGit2 engine, but none of Git engines are faster than the old one since they take some time to compress files. Figure 5 also shows a boxplot describing the average execution time of the four executions for each content database engine. Notice, the PyGit2 is much faster than the Dulwich engine but has almost the same effect in reducing the storage size, when compared to the old engine.

We also applied hypothesis tests to the time generated data set of the Old and PyGit2 version, so we can confirm that our changes affect the time negatively. First, we applied the Shapiro Wilk test (SHAPIRO; WILK, 1965) to check the normality. Since both distributions are not normal we have to use a non-parametric test to compare the data means, the Wilcoxon Signed-rank Test (WILCOXON, 1945). Our null hypothesis again is that the averages are equal. The p-value found was equal to $1.563e-2$, bellow 0.05, meaning that the averages of time are non-identical and that our changes affect the time.

We also applied a non-parametric effect size test Cliff's Delta (CLIFF, 1993). The resulting delta is $-1.428571e-1$ which is considerable negligible. It means that we apply effect on the execution time, but that time is negligible.

Git Garbage Collection. We created a synthetic experiment because the previous ones are already finished, so we cannot see the evolution of an experiment workstation and how that scales in term of size. The experiment we create simulates a trial-and-error process that appends to a text file a random text content (around 13 Megabytes) in every execution. This simulates multiple executions of an experiment that grows the size of a file over the executions. We execute this script 10 times, which increases a specific file in each trial. After the 10 executions, we execute the *now gc* command that delegates to the Git

installed on the machine the execution of the garbage collection over the *content database* repository. Table 5 shows the size evolution of the file over the 10 executions and after that the execution of the *now gc* command. Notice that the size of Git engines is always lower than the old one. At the GC column, which executes the garbage collection tool, the size of the *content database* directory with the Git engines reduces 73.79%, while the old engine does not support this feature.

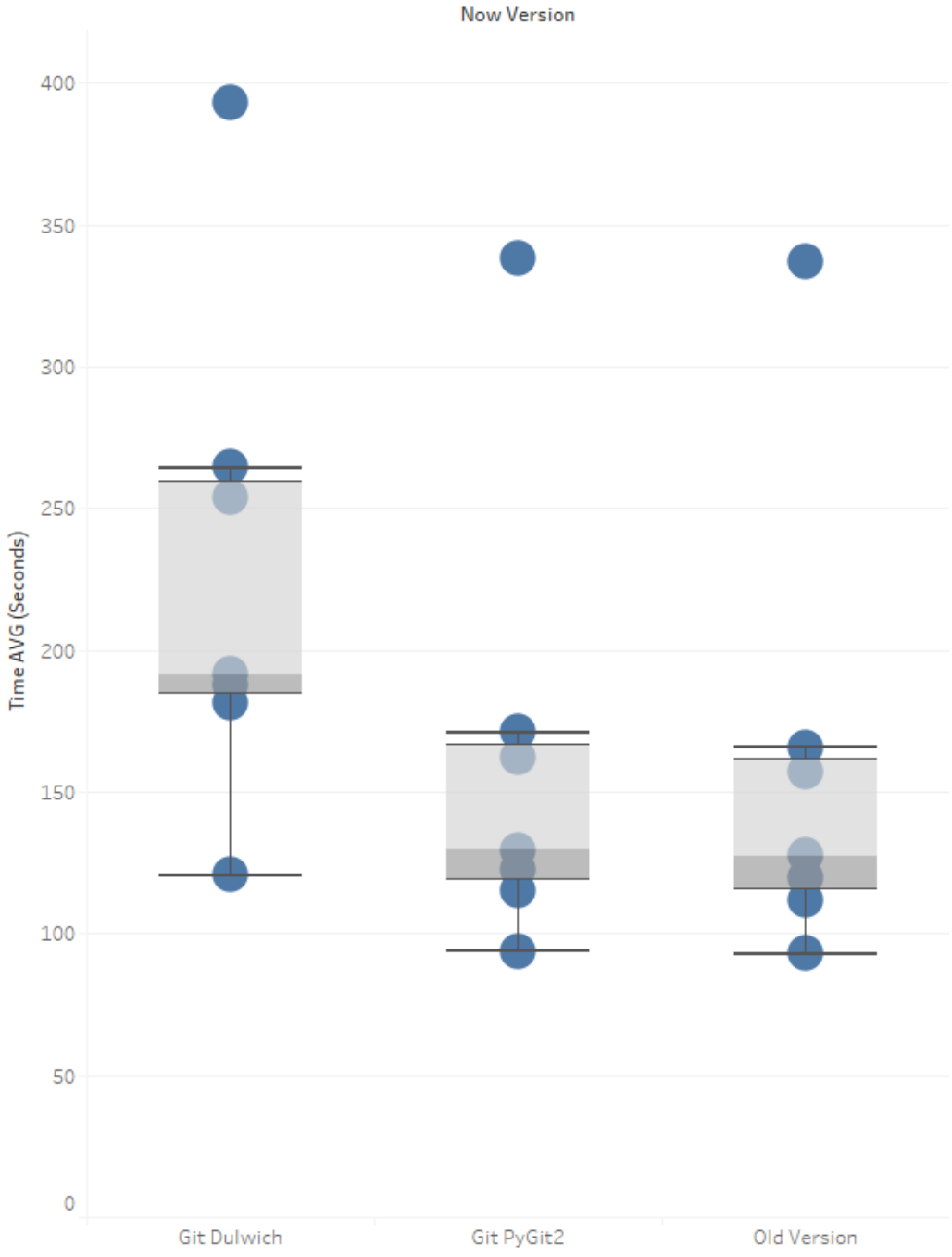

Figure 5 – Elapsed time of all scripts executions for each content database engine version.

Table 5 – Size in Megabytes of content database directory for every 10 executions of noWorkflow with the research simulation script and the execution of *now gc* at the final.

Now Version	Trial - 01	Trial - 02	Trial - 03	Trial - 04	Trial - 05	Trial - 06	Trial - 07	Trial - 08	Trial - 09	Trial - 10	GC
Git Dulwich	12.2	32	61.6	101.1	150.4	209.6	278.7	357.6	446.4	545.1	189.1
Git PyGit2	12.6	32.8	63	103.4	153.8	214.3	284.9	365.6	456.3	557.1	189.1
Old Version	18.1	44.2	83.3	135.3	200.5	278.6	369.7	473.9	591.1	721.3	721.3

Evaluation Results. The results show that the Git engines reduce the size overhead of those experiments with a tiny processing time overhead. Moreover, the Dulwich execution time is 43.32% slower than the old engine on average. On the other hand, the PyGit2 library is just 1.90% slower on average. We believe the difference is because the core of PyGit2 is built in C while the Dulwich is completely written in Python. Moreover, we tried to do system calls with the Git installed in the user's machine, but it has proved inefficient due to the time overhead added. For this reason, we set the PyGit2 as our main content database engine. The garbage collection tool reduced the content database size in 73.79%. We believe this reduction is this high because we are dealing with text files which can achieve a good compression. If we have manipulated other file formats the compression could not be so high. We did not execute the garbage collection tool in all experiments because there would be no significative reduction of size since garbage collection usually benefits the most from multiple versions of the same file.

5 RELATED WORK

There are many approaches that capture provenance from scripts and each of them stores provenance in different manners. Some approaches do not capture the content of files and only store the provenance data in the system memory (RUNNALLS, 2011), databases (BOCHNER *et al.*, 2008; MWEBAZE *et al.*, 2009), or OPM files (TARIQ; ALI; GEHANI) which does not consider the content of files. Thereby, they do not suffer the storage problems we are trying to solve.

Datatrack (EICHINSKI; ROE, 2016) and Sumatra (DAVISON, 2012) use version control systems to store the provenance. Although, these approaches use all the features of version control systems in the porcelain API, inheriting some of their limitations. In our case, we use the file storage tools of Git at the plumbing level. It allows us to store many versions of a given file within a single trial version.

ESSW (FREW; BOSE, 2001), IncPy (GUO; ENGLER, 2011) and versuchung (DIETRICH; LOHMANN, 2015) use content databases to store files as well. However, they use different ways to identify the files. They could benefit from our approach with some adjustments.

6 CONCLUSIONS

We have presented an alternative storage mechanism to the noWorkflow content database, which applies Git to reduce the storage size overhead. By using Git as a file database, we reduce the size overhead by 65.23% on average, with an extra processing time overhead of 1.90% on average. Moreover, in our synthetic experiment, the size of the repository reduces by 73.79% after the garbage collection tool execution. Additionally, this integration automatically creates Git commits timeline in the repository along with executions.

As the Git libraries do not implement all Git functionalities, like the garbage collection command, to fully use the new storage mechanism, one should have Git installed in the user machine. However, even without Git, most of the noWorkflow functionalities are still available.

We believe that Git has many other powerful tools to apply to noWorkflow. A future work could be using Git to enable users to create branches, allowing them to separate different trials through it (NEVES *et al.*, 2017). Another future work is to use Git as an additional form of provenance analysis by providing an interface that the users could do more detailed diff analysis. Finally, we envision using Git for publishing and accessing provenance collected by noWorkflow to enhance collaboration among scientists.

7 REFERENCES

ADLER, Mark. *zlib: A massively spiffy yet delicately unobtrusive compression library*. Available: <https://github.com/madler/zlib>. Accessed: 13 jun. 2018.

BOCHNER, Carsten; GUDE, Roland; SCHREIBER, Andreas. A Python Library for Provenance Recording and Querying. In: INTERNATIONAL PROVENANCE AND ANNOTATION WORKSHOP, Lecture Notes in Computer Science, 17 jun. 2008, [S.l.]: Springer, Berlin, Heidelberg, 17 jun. 2008. p. 229–240. Available: <https://link.springer.com/chapter/10.1007/978-3-540-89965-5_24>. Accessed: 25 jun. 2018.

CHACON, Scott; STRAUB, Ben. *Pro Git*. 2. ed. [S.l.]: Apress, 2014. Available: <https://github.com/progit/progit2/releases/download/2.1.67/progit.pdf>.

CLIFF, Norman. Dominance Statistics: Ordinal Analyses to Answer Ordinal Questions. *Psychological Bulletin*, v. 114, p. 494–509, 1 nov. 1993.

COOK, John. *Code to slice open a Menger sponge*. Available: <https://www.johndcook.com/blog/2011/08/30/slice-a-menger-sponge/>. Accessed: 15 jun. 2018.

DANG, Quynh H. *Secure Hash Standard*. , nº NIST FIPS 180-4. [S.l.]: National Institute of Standards and Technology, jul. 2015. Available: <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>. Accessed: 17 jul. 2018.

DAVISON, A. Automated Capture of Experiment Context for Easier Reproducibility in Computational Research. *Computing in Science Engineering*, v. 14, n. 4, p. 48–56, jul. 2012.

DEY, Saumen; BELHAJJAME, Khalid; KOOP, David; RAUL, Meghan; LUDASCHER, Bertram. Linking Prospective and Retrospective Provenance in Scripts. p. 7, 2015.

DIETRICH, Christian; LOHMANN, Daniel. The Dataref Versuchung: Saving Time Through Better Internal Repeatability. *SIGOPS Oper. Syst. Rev.*, v. 49, n. 1, p. 51–60, jan. 2015.

dulwich: Pure-Python Git implementation. Available: <https://github.com/dulwich/dulwich>. Accessed: 13 jun. 2018.

EICHINSKI, P.; ROE, P. Datatrack: An R package for managing data in a multi-stage experimental workflow data versioning and provenance considerations in interactive scripting. In: 2016 IEEE 12TH INTERNATIONAL CONFERENCE ON E-SCIENCE (E-SCIENCE), oct. 2016, [S.l: s.n.], oct. 2016. p. 147–154.

FREW, J.; BOSE, R. Earth System Science Workbench: a data management infrastructure for earth science products. In: PROCEEDINGS THIRTEENTH INTERNATIONAL CONFERENCE ON SCIENTIFIC AND STATISTICAL DATABASE MANAGEMENT. SSDBM 2001, 2001, [S.l: s.n.], 2001. p. 180–189.

GILL, Richard D. Event based simulation of an EPR-B experiment by local hidden variables: epr-simple and epr-clocked. *arXiv:1507.00106 [quant-ph]*, arXiv: 1507.00106, 1 jul. 2015. Available: <http://arxiv.org/abs/1507.00106>. Accessed: 15 jun. 2018.

GUO, Philip J.; ENGLER, Dawson. Using Automatic Persistent Memoization to Facilitate Data Analysis Scripting. ISSTA '11, 2011, New York, NY, USA. *Anais*... New York, NY, USA: ACM, 2011. p. 287–297. Available: <http://doi.acm.org/10.1145/2001420.2001455>. Accessed: 25 jun. 2018.

HEDGES, Larry V.; OLKIN, Ingram. *Statistical Methods for Meta-Analysis*. 1 edition ed. Orlando: Academic Press, 1985.

librosa: Python library for audio and music analysis. Available: <https://github.com/librosa/librosa>. Accessed: 15 jun. 2018.

MURTA, Leonardo; BRAGANHOLO, Vanessa; CHIRIGATI, Fernando; KOOP, David; FREIRE, Juliana. noWorkflow: Capturing and Analyzing Provenance of Scripts. In: LUDÄSCHER, BERTRAM; PLALE, BETH (Org.). . *Provenance and Annotation of Data and Processes*. Cham: Springer International Publishing, 2015. v. 8628. p. 71–83. Available: <http://link.springer.com/10.1007/978-3-319-16462-5_6>. Accessed: 9 jun. 2018.

MWEBAZE, J.; BOXHOORN, D.; VALENTIJN, E. Astro-WISE: Tracing and Using Lineage for Scientific Data Processing. In: 2009 INTERNATIONAL CONFERENCE ON NETWORK-BASED INFORMATION SYSTEMS, aug. 2009, [S.l: s.n.], aug. 2009. p. 475–480.

NEVES, Vitor C.; OLIVEIRA, Daniel De; OCAÑA, Kary A. C. S.; BRAGANHOLO, Vanessa; MURTA, Leonardo. Managing Provenance of Implicit Data Flows in Scientific Experiments. *ACM Trans. Internet Technol.*, v. 17, n. 4, p. 36:1–36:22, aug. 2017.

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, 1 aug. 2017.

pygit2: Python bindings for libgit2. Available: <https://github.com/libgit2/pygit2>. Accessed: 13 jun. 2018.

RUNNALLS, Andrew R. Aspects of CXXR internals. *Computational Statistics*, v. 26, n. 3, p. 427–442, 1 sep. 2011.

SHAPIRO, S. S.; WILK, M. B. An analysis of variance test for normality (complete samples). *Biometrika*, v. 52, n. 3–4, p. 591–611, 1 dec. 1965.

TARIQ, Dawood; ALI, Maisem; GEHANI, Ashish. Towards Automated Collection of Application-Level Data Provenance. p. 5, [S.d.].

THE PROBABLE ERROR OF A MEAN. *Biometrika*, v. 6, n. 1, p. 1–25, 1 mar. 1908.

WICKERT, A. D. Open-source modular solutions for flexural isostasy: gFlex v1.0. *Geosci. Model Dev.*, v. 9, n. 3, p. 997–1017, 8 mar. 2016.

WILCOXON, Frank. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, v. 1, n. 6, p. 80–83, 1945.

WINCKEL, Greg Von. *qho-2-electrons: Quantum harmonic oscillator with two interacting electrons*. Available: <https://github.com/gregvw/qho-2-electrons>. Accessed: 15 jun. 2018.