Exploring Existing Tools for Managing Different Types of Research Data

Adrian Freund [1], Hamideh Hajiabadi [2], and Anne Koziolek [2]

Abstract: Data management is important for the reproducibility of scientific research. One important aspect of data management is version control. In software development, version control tools like Git are commonly used to track source code changes and releases, reproduce earlier versions, find defects, and simplify their repair. In scientific research, scientists often have to manage large amounts of data, while also trying to achieve reproducibility of results and wanting to identify and repair defects in the data. Version control software like Git is specialized for managing source code and other textual files, making it often unsuitable for managing other types of data. This creates a need for version control tools specialized for dealing with research data. This paper establishes requirements for version control tools for research data and evaluates Git Large File Storage, Neptune, Pachyderm, DVC, and Snowflake according to those requirements. We found that none of the evaluated tools fulfill all of our requirements, but we still recommend DVC, Git LFS, and Pachyderm for the use cases they do support.

Keywords: Data management, Version control, Reproducibility, FRBR model

1 Introduction

Scientific research often involves collecting and managing large amounts of data and drawing conclusions from the collected data. During an ongoing research project, datasets might change as new data is collected, outdated data is removed, or wrong data is corrected. This makes reproducing previous results challenging, as the dataset used to obtain the original result might no longer be available or might not be able to be identified. This makes data management a necessity for reproducibility in research [BV22]. A critical aspect of data management is version control [KI21].

Some software used to analyze the datasets is custom-written for a specific research project and might also be updated with new features, improved analysis, and fixed bugs during the research. This makes it necessary to version not only the datasets used but also the software used to analyze them. In some cases, changes in the underlying operating system and used libraries can cause changes in the result.

In addition to this, research datasets are getting more significant. In some cases, downloading datasets is no longer feasible due to their size [KI20]. With more and more research datasets

[1] Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, Karlsruhe, Germany, adrian.freund@student.kit.edu, https://orcid.org/0009-0002-5220-1042

[2] Karlsruhe Institute of Technology (KIT), Institute of Information Security and Dependability (KASTEL), Am Fasanengarten 5, Karlsruhe, Germany, hamideh.hajiabadi@kit.edu, https://orcid.org/0000-0002-5793-4563; koziolek@kit.edu, https://orcid.org/0000-0002-1593-3394

doi:10.18420/inf2024_189

becoming available online instead of being created by researchers, it becomes even more important to introduce systematic data versioning, allowing researchers to cite the data used [KI20] correctly.

Reproducibility can be divided into two concepts: internal reproducibility, which is the ability of a research team to reproduce their own research, and external reproducibility, which is the ability of other researchers to reproduce research once it is published. External reproducibility requires researchers to publish their findings and their data in ways usable for other researchers [BV22]. A significant factor in researchers not publishing their data in such a way is that it is not part of a team's regular workflow and is seen as *extra work* [BV22]. Better data management tools, including version control, could simplify this process, making it easier and less time-consuming to publish usable data and, therefore, allowing better external reproducibility. While source code is usually textual, research data can be represented in various formats, including textual formats such as CSV, binary formats, or databases. A version control system for research data should efficiently handle different data types.

This paper evaluates multiple research data versioning systems' ability to version and manage research data. We will first take a closer look at what types of data researchers typically work with. After that, we will discuss concepts for version-controlling research data. In the central part, we will examine the research data management systems Git LFS, Neptune, Pachyderm, DVC, and Snowflake more closely and evaluate their features and adherence to those principles.

2 Related Work

Seinhorst and Boekhoudt already evaluated data versioning tools DataHub and OrpheusDB and compared them to both an Ad Hoc data versioning process and a simple database without versioning features [SB19]. This paper will focus on other tools.

Seering et al. developed a versioning system using *named versions* and branches for the scientific array database SciDB in 2012 [Se12]. The SciDB website linked in that paper is no longer available and web searches only show results for other software with the same name. Because of this SciDB could not be evaluated in this paper.

Schüle et al. presented TardisDB [Sc21]. TardisDB is a relational database system with built-in support for data versioning using branches [Sc21]. TardisDB is Open Source and its source code is available on GitHub[3]. There are currently no binary distributions of it. TardisDB extends Structured Query Language (SQL) with a new `VERSION` keyword that allows retrieving, updating, and inserting data from and into specific versions of a database table [Sc21]. While this functionality looks promising, TardisDB is a research prototype and not suitable for use in production systems, so it was also not evaluated in this paper.

[3] https://github.com/tum-db/TardisDB

eSciDoc is an Open Source e-research environment developed by Max Planck Digital Library and FIZ Karlsruhe. It has support for versioning both *working versions* and *releases*, as well as collections, that can contain multiple, independently versioned objects [Ra09]. Development of eSciDoc stopped in 2012 and its authors don't recommend its use for new projects due to security issues. For this reason, eSciDoc will also not be evaluated in this paper.

Zenodo[4] is a data publishing platform commissioned by the European Commission and developed by CERN. While it has the ability to publish different versioned by specifying a version string on every upload, it does not provide any tools to help with the versioning. Because of this it will also not be included in this evaluation.

In a developer survey[5], it was reported that version control systems such as Git and SVN are already common in software development. At the time of writing, the most used version control system among professional developers is Git, which is the primary version control system for 96.65% of professional developers. Additionally, 98.62% of professional developers and 82.82% of beginner programmers use some kind of version control system. In software development, version control is used to keep versions of software, source code, and configurations organized [Ti85].

While Git is good at version-controlling code, it has some shortcomings, that hinder its usefulness for versioning research data. Git is very inefficient when dealing with large files [Ma15].

Git is a command-line tool that supports local and remote repositories. Multiple Git server implementations can be used to host a remote repository. Git servers like GitHub and GitLab also provide a web interface for a subset of Git's functionality [Br18]. When working with a git repository, one first has to clone the repository. Cloning a Git repository[6] with default options downloads the whole repository history, including all historic data. It is possible only to download the current version of a Git repository using a *shallow clone*. This, however, severely limits Git's functionality on this repo, preventing the uploading of changed versions back to the server. For datasets where a single revision is already too large to download Git isn't useful at all.

3 Version Control Concepts

Now that we have established the types of data a version control system for research data must handle, we will examine the functionality expected of such a system.

If we want to version research data we first need to identify what a version is. For this, we will use a version of the *Functional Requirements for Bibliographic Records* (FRBR) model,

[4] https://about.zenodo.org/
[5] https://survey.stackoverflow.co/2022
[6] https://www.peterlundgren.com/blog/on-gits-shortcomings

developed by the International Federation of Library Associations and Institutions (IFLA). The FRBR was originally developed for to describe how information resources relate to each other in the context of bibliographic records.

Based on this model and 39 collected use cases Klump et al. first defined five principles for data versioning [KI20] and later added a sixth principle [KI21].

Version Control and revisions All new dataset instances constitute a new *revision* [KI20]. Changes to the metadata of a dataset do not constitute a new revision [KI20]. Every new revision needs to be identified. This can be done by creating a new, persistent, and unique identifier for that release [KI21].

Identifiying releases of a data product A dataset might undergo multiple revisions before a revision is designated as a *release* [KI20]. There are no clear guidelines for when a revision should be designated as a release. Klump et al. describe a release as a revision considered *final* [KI21].

A description of the release must accompany a release. This should include information about the release, its nature, and the significance of the changes since the last release. This includes information about possible incompatibilities between the old and new releases, requiring users to adjust their processes when using the dataset. *Semantic Versioning* can be used to efficiently communicate the significance of the change and degree of compatibility [KI20].

Identification of data collections Some datasets might be collections of other datasets, such as a time series. When a dataset consists of other datasets, the overall and all contained datasets must be independently versioned. The overall dataset is revised when a contained dataset is changed to a different revision.

Identification of manifestations of datasets One dataset can be distributed as many different manifestations [KI21], for example, with different encodings or different structures. While different manifestations of the same expression contain the same information, there might be technical considerations for identifying them seperately [KI21]

Requirements for provenance of datasets *Provenance*, in this context, describes the knowledge of the history of a piece of information, as well as its origin [Ta15]. It includes information on how a piece of information changed and where it originates so its trustworthiness can be established [Ta15]. To enable those use cases a dataset release should include information on its provenance [KI21]. This includes both revision histories for datasets and information about how and from what a derived dataset was derived.

Requirements for data citation When published, they should be accompanied by metadata that allows easy citation. The DataCite metadata[7] kernel gives requirements and recommendations for what information should accompany the publication. This includes a

[7] https://datacite-metadata-schema.readthedocs.io/en/4.5/

persistent identifier, a title, the dataset's creators and publishers, and the publication date. This principle is especially important for external reproducibility.

Therefore, a version control system for research data should be able to help users apply those principles to their datasets. Whenever possible it should enforce principles to prevent misuse.

4 Existing Systems for Version Control of Research Data

In this section, we will look closely at some data management tools and evaluate their usefulness for version-controlling research data. The goal of this paper is not to give a comprehensive overview of all data version control systems. Instead, we will look closer at Git FLS, Neptune, Pachyderm, DVC, and Snowflake. The tools will be evaluated on their technical ability to deal with large and very large datasets and their ability to version data according to the six principles discussed above.

The evaluation in this section is based on available documentation and our usage of the software in its free version, where available, or a time-limited trial otherwise. In addition, some information about Pachyderm comes from a Hewlett Packard Enterprise (HPE) representative we have been in contact with and a prerecorded software demo they provided us with.

4.1 Git LFS

Git LFS[8] is an extension for Git that improves handling large files. Multiple server implementations, such as GitHub, GitLab, and Gitea, are available. GitHub offers 1 GB of Git LFS storage on its free and paid plan, with more storage available for additional payment. The maximum file size limit for the Git LFS implementation by GitHub varies between 2 GB and 5 GB, depending on the payment plan.

Git LFS replaces large files inside the Git repository with references to files stored outside of the repository, which allows the actual files to be retrieved on demand. It can be activated by using the `git lfs track <file_pattern>` command to track files using Git LFS.

Git LFS replaces tracked files with references and uploads the actual files to a Git Large File Store. When cloning a repository using Git LFS, only revisions of tracked files used in the current version are downloaded, resulting in a significant speedup compared to normal Git.

However, downloading a full revision of a large dataset might not be feasible, making it difficult to manage large or very large datasets using Git LFS. Therefore, Git LFS is only feasible for managing small to medium-sized datasets.

[8] https://git-lfs.com

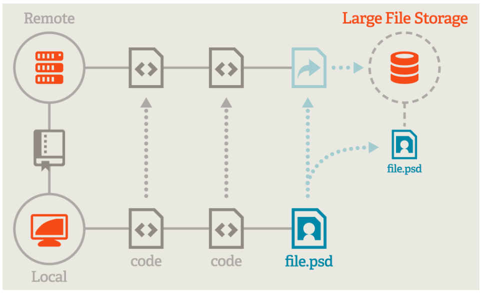

Fig. 1: A diagram from the Git LFS Website visualizing how Git LFS replaces large files with references

Git LFS can potentially align with most of Klump et al.'s principles for version control. As it is based on Git, it can benefit from most of its features. Versioning of datasets can be represented as Git commits, which save the current state of the repository along with some metadata, including the author, date, and time, a free text commit message, and an electronic signature [CS14]. Each commit also receives a persistent identifier called a commit ID. Git commit IDs are cryptographic hashes of the content of the commit, its metadata, and a reference to the previous commit in the history [CS14]. This makes them completely deterministic, but due to the cryptographic hash, it is still very likely to be unique.

The release of dataset versions can be represented as Git tags. Git provides two types of tags: lightweight and annotated tags. A lightweight tag is a named reference to a specific revision. An annotated tag includes an additional annotation message and free text metadata added to the tag [CS14]. This annotation message field can be used to describe the nature of the release and its significance. However, it is not mandatory, allowing for the release of datasets without adequate release information. While the tag names can be used for a Semantic Versioning version number, it is not enforced, allowing for arbitrary identifiers. Although Git does not natively support adding metadata files or structured data to tags, some Git servers, such as GitHub and GitLab, add them as an additional feature on top of Git.

Git allows fine-grained access to specific revisions of single files in a repository. The command `git checkout <revision> -- <file>` makes it possible to retrieve a file version from an older revision without loading the whole dataset. Git also supports submodules, which are pointers directing to a specific revision of a different Git repository. This allows for versioning of dataset collections. Changing the revision of a submodule requires creating a new Git commit and producing a new revision of the containing repository.

Git does not support identifying different manifestations of a dataset. As commit IDs are deterministically dependent on the content of a repository, a repository containing a different manifestation will have different commit IDs.

Provenance is partially supported. One can view the full revision history of a Git repository, enabling the user to see when changes were made and by whom. Git provides the `git blame` command to determine in which revision a file or a part of a file was last changed. However, information regarding the origin of the added data must be entered as free text in the commit message field, which limits users from adequately describing the data's provenance.

Datasets stored in Git LFS can be published by storing the relevant Git repository on a Git server and setting it to public. However, publicizing the repository allows access to the published release and the complete history. The only way to add additional citation metadata is as a file in the repository. If someone wants to publish a specific release only, they must first export it from Git LFS.

Additionally, Git has the advantage of being widely used in software development, making it easier to manage datasets with the code used to analyze them.

4.2 Neptune

Neptune[9] is a commercial, proprietary product for tracking machine learning data, metadata, and models and is developed by Neptune Labs, Neptune specializes in tracking explorative machine learning work. It is available both as a free and paid cloud service. The enterprise version is also available for self-hosting.

Neptune[10] provides a Python and R library that can be integrated into training programs to upload and track data on every program run automatically. As most functionality is only documented for the Python library this evaluation will also focus on the Python library. Some data, such as general system information and the running program's source code, are tracked automatically on every run just by initializing the library. In contrast, the training program must track other data using library calls. When initializing the Neptune library one gets provided with a `Run` object. This `Run` object can be used to track data. The `upload` method can be used to upload files, while a simple assignment can be used to track plain values. Neptune can also integrate with some popular machine learning libraries to automatically track relevant data, requiring less integration work by the developer.

Uploading large datasets to the server on every run is impractical. Neptune doesn't specify if it can detect unchanged data automatically. We tested this and found that even unchanged data is uploaded on subsequent runs. To deal with large datasets, Neptune provides `track_files` method to upload only file metadata instead of content. However, this approach has limited usefulness for reproducibility since it only identifies the exact dataset used for a run. Reproducing the results requires obtaining the dataset from another source.

The web interface compares graphs, output images, and metadata between different runs. It is also possible to create and manage model releases.

[9] https://neptune.ai
[10] https://docs.neptune.ai/

Because Neptune uploads and stores relevant data on every training run, tracking revisions is automatic. A persistent identifier for a revision can be either specified manually or, if left blank, generated by incrementing an integer by one for every run. As Neptune needs to be manually integrated into one's programs, it is possible to forget to track relevant data accidentally. Also, in cases where models are trained infrequently, many changes to the dataset might happen between runs, causing very large differences between versions.

Neptune only supports creating releases for trained models. Releases for the input dataset used to train the model are not possible.

Because Neptune has no concept of datasets, only files and directories, a collection of datasets can only be represented as a directory. It is not possible to version contained datasets independently from the containing dataset. Only the dataset as a whole will be versioned. It is, however, possible to retrieve individual files of older revisions without downloading the whole revision using the `download` method on the `Run` object.

There is also no support for identifying different manifestations of a single dataset.

Support for data provenance is also limited. While it is possible to see a history of runs, the data stored in those recorded runs is just whatever the current state of the dataset was at runtime, without a way to track where changes have come from. Because derived data, such as trained models, can be stored together with both the datasets and the source code used to train them, provenance for derived data can be established, given that Neptune was used to derive the datasets.

Projects on Neptune can be published, allowing anyone to view but not change them. It should be noted that this allows access to all project data, including all revisions, derived data, and metadata. Neptune also supports adding structured metadata to projects, making it possible to add citation metadata. There is, however, no predefined scheme for how such metadata should be structured.

4.3 Pachyderm

Pachyderm[11] is a data-pipeline automation software owned by Hewlett Packard Enterprise (HPE). It is available as both a paid enterprise edition and a free Open Source community edition. According to an HPE representative, the enterprise edition has recently been renamed to HPE Machine Learning Data Management Platform, although this change is not yet reflected on the Pachyderm website. The community edition has constraints on the number of pipelines and parallel workers, while the enterprise version is limited only by hardware resources. The enterprise edition also supports more customizable authentication for enterprise settings. For this evaluation, the community edition was used.

[11] https://www.pachyderm.com, https://docs.pachyderm.com/, https://www.youtube.com/watch?v=BcKixx4Rhbw

Pachyderm is available only as a self-hosted server. It aims to provide a comprehensive data-analysis environment, including data storage, versioning, task and pipeline management, execution, and output storage and versioning. Data is stored as files and directories within data repositories. Projects can have multiple pipelines, each connecting one or more input repositories to an output repository, which can then be used as an input repository for other stages. Tasks are executed using container images dispatched to a Kubernetes cluster, enhancing reproducibility and allowing users to utilize various programming languages and tools. Pipelines can distribute datasets across multiple containers, enabling efficient scaling with large datasets.

All code runs on the Kubernetes cluster, allowing less powerful computers to work without downloading all the data. Users can interact with Pachyderm via a web interface, the `pachctl` command-line tool, and a Python package.

Pachyderm supports versioning for input and output data repositories, similar to Git repositories. Files are automatically deduplicated to conserve storage. Existing pipelines automatically rerun upon changes in input repositories, with outputs committed to corresponding repositories. Pipelines can be run against old commits to reproduce results. Revisions are tracked automatically for changes in data repositories. Multiple files can be added with one commit using commands like `pachctl put file` and `pachctl start/finish commit`, and Pachyderm generates persistent commit IDs as identifiers.

However, Pachyderm lacks support for adding tags to commits and referencing other repositories, limiting proper versioning of dataset collections. It supports provenance through commit messages and automatic versioning of output data linked to pipelines and input repositories. The community version lacks advanced authentication features, which limits its data publication capabilities. Despite these limitations, Pachyderm is a robust option for versioning research data.

4.4 DVC

Data Version Control[12] (DVC) is an Open Source data and data pipeline management tool developed by iterative.ai. An enterprise version called DVC Studio is also available. While the open-source version offers only a command-line tool, the enterprise version offers a web interface with additional functionality. This evaluation uses the Open Source version. DVC is designed for machine learning and supports data, experiment, and model management. In this evaluation, we will only look at the data management functionality.

DVC is designed to be used together with Git. In contrast to Git LFS, it does not directly integrate with Git as an extension. Instead, DVC lets its metadata be versioned in a Git repository. Like Git LFS, DVC keeps large files from the Git object database by storing them separately. DVC does however not require support for this from the Git server. Instead,

[12] https://dvc.org

the user can provide DVC with a remote server. S3 object stores, a server with SSH access, other cloud storage providers, and more are supported. When DVC is told to track a file called `file_name` it adds an additional file called `file_name.dvc` containing metadata about the file. Only this additional metadata file gets added to the Git object database. The original file is automatically added to the `.gitignore` file to prevent it from getting also added. Using `dvc push` the original file can be uploaded to the configured remote server. When a user obtains a repository with only the metadata file he can run `dvc pull` to retrieve the original file from the remote server.

In addition to managing large files, DVC can also manage data pipelines and the derived datasets they produce. A pipeline can have multiple stages, where each stage uses one or more files from the repository as inputs and creates one or more files or directories in the repository as outputs. DVC automatically tracks pipeline outputs as large files. Because all inputs, pipelines, and outputs are stored in the same Git repository, they can be versioned together. In contrast to Pachyderm, DVC does not automatically rerun pipelines when input data changes, but only when the user requests it to use `dvc repro`. When asked to rerun pipelines, it checks which data has changed and only runs the necessary pipeline stages. While Pachyderm supports sharding data and running pipelines on a Kubernetes cluster DVC pipelines always run locally.

Because all DVC metadata and states are stored in a Git repository, the same Git features as those of Git LFS can be used. Git commits can represent dataset revisions with a commit ID as a persistent identifier. Releases can be tagged with Git, and additional information about the release is stored as free text in an annotated tag. DVC can also be used with Git submodules to support versioning of dataset collections; however, DVC commands like `dvc push` and `dvc pull` must be manually run from inside. DVC also doesn't add any support for identifying different manifestations of datasets. However, it improves provenance support by adding data pipelines that track their in and output datasets and establish a provenance between them. Support for publishing and data citing is again the same as with Git LFS, with the additional requirement of making the remote storage server available to other users. This is not necessary with Git LFS, as the server is built into the Git server.

DVC provides roughly the same versioning functionally as Git LFS and adds data pipelines. It does, however, not integrate into Git as tightly, often requiring users to run two commands (one `git` and one `dvc` command) to accomplish what Git LFS can do with one command.

4.5 Snowflake

Snowflake[13] is a leading commercial cloud database tailored for managing extensive datasets. Accessible via SQL and a dedicated Python library, it offers unique SQL extensions, including the Time Travel extension for historical data retrieval. However, its historical data retention is limited to 90 days, posing challenges for reproducing earlier research beyond this timeframe.

[13] https://www.snowflake.com, https://docs.snowflake.com/

Automatic revision tracking in Snowflake, utilizing timestamps or query IDs, provides granular control over dataset changes. Yet, query IDs are valid for just 14 days, imposing constraints on historical data access. While lacking a native release management framework, Snowflake supports snapshotting through lightweight clones, allowing users to manually manage release-related information.

Snowflake databases lack inter-database referencing, complicating comprehensive versioning of dataset collections. While not inherently supporting the identification of dataset manifestations, its use of explicit schemas aids dataset comprehension. Provenance tracking in Snowflake is modest, with a 7-day query history retention period and no provision for adding metadata to queries. The Snowflake Marketplace enables dataset publication and description, albeit restricted to Snowflake users.

5 Discussion

In this paper, we explored what properties a version control system for research data should have and looked at five systems used in the real world. While none of the analyzed systems manage to satisfy all of the requirements we established they can all be useful for a subset of our goals. The results have been summarized in Table 1

	Git LFS	Neptune	Pachyderm	DVC	Snowflake
Supports very large datasets			x		x
Revisions	x	-	x	x	-
Releases	x	-	-	x	-
Collections & Granularity	x			x	
Manifestations					-
Provenance	-		x	x	
Publishing & Citation	-	-		-	x

Tab. 1: Comparison of features across various version control systems. The table highlights the support for different versioning principles, such as handling very large datasets, revisions, releases, collection and granularity management, manifestations, provenance tracking, and publishing and citation capabilities in Git LFS, Neptune, Pachyderm, DVC, and Snowflake. An "xïndicates that the feature is supported, while a ïndicates that the feature is partially supported. When it is blank, the feature is not supported at all.

DVC satisfies the most requirements. It fully satisfies four requirements and partially supports an additional requirement. Is it, however, limited by not supporting very large datasets too large to download?

Both Git LFS and Pachyderm fully satisfy three requirements. Git LFS additionally partially satisfies two more requirements, while Pachyderm only partially satisfies one additional requirement. However, the workflows when working with those two systems are very different. The biggest weakness of Git LFS is its inability to deal with large datasets. Popular Git LFS servers limit filesize and Git LFS does not provide any tooling to work with datasets without downloading them first, making its use with datasets that are too large to download impossible. Git LFS also only provides versioning for source code and data. Pachyderm, on the other hand, provides a much more complete platform that deals with data version control and tools for building data analysis pipelines and versioning derived data, with analysis pipelines running on a server or even cluster. Of course, Git LFS can also store version-derived data, but provenance must be manually established. Git LFS, on the other hand, is better equipped to handle dataset releases. Git tags can identify a revision as a release and describe it using free text metadata. Pachyderm only provides limited support for dataset releases using repository branches.

Git LFS and DVC are similar, but Git LFS provides a better user experience due to its tight integration with Git. Neptune is specialized for explorative machine learning and lacks general version control features. Researchers using Neptune might also use Git LFS to version their data.

Snowflake was the only database in this analysis. While Snowflake has some functionality for retrieving older data revisions, it is unsuitable for version control, as its version history is limited to 90 days at most. The maximum history might be as short as 7 days for some use cases. Dataset releases could still be implemented in an ad hoc process using database snapshots. Notably, Snowflake was the only tool we evaluated that has specialized tooling for data publishing.

6 Conclusion

This paper highlights the importance of data management and version control in research to improve reproducibility. We analyze different research data types and the necessary criteria for these tools. Based on these criteria, we then evaluate the version control features of Git LFS, Neptune, Pachyderm, DVC, and Snowflake.

None of the tools fully meet all our requirements. However, Git LFS, Pachyderm, and DVC are recommended for managing research data, with the choice depending on the specific project. Database version control is still developing, with tools like TardisDB showing promise but not yet suitable for production.

Acknowledgement

This work was created as part of the NFDI consortium NFDIxCS (nfdixcs.org) and is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under the National Research Data Infrastructure—NFDI 52/1—project number 501930651.

Bibliography

[Br18] Bryan, J.: Excuse Me, Do You Have a Moment to Talk About Version Control? The American Statistician 72(1), pp. 20–27,issn: 0003-1305, 1537-2731, doi: 10.1080/00031305.2017.1399928, 2018.

[BV22] Borghi, J.; Van Gulick, A.: Promoting Open Science Through Research Data Management. Harvard Data Science Review, doi: 10.1162/99608f92.9497f68e, 2022.

[CS14] Chacon, S.; Straub, B.: Git Internals. In: Pro Git. 2nd Edition, Apress, pp. 414–452, isbn: 978-1-4842-0077-3, 2014.

[Kl20] Klump, J. et al.: Principles and Best Practices in Data Versioning for All Data Sets Big and Small. doi: 10.15497/RDA00042, 2020.

[Kl21] Klump, J. et al.: Versioning Data Is About More than Revisions: A Conceptual Framework and Proposed Principles. Data Science Journal 20, p. 12, issn: 1683-1470, doi: 10.5334/dsj-2021-012, 2021.

[Ma15] Madden, S. R. et al.: DataHub: Collaborative Data Science & Dataset Version Management at Scale. Proceedings of the 7th Biennial Conference on Innovative Data Systems Research, 2015.

[Ra09] Razum, M. et al.: e-SciDoc Infrastructure: A Fedora-Based e-Research Framework. In: Research and Advanced Technology for Digital Libraries. Vol. 5714, Springer Berlin Heidelberg, pp. 227–238, isbn: 978-3-642-04345-1, doi: 10.1007/978-3-642-04346-8_23, 2009.

[SB19] Seinhorst, T. C.; Boekhoudt, K.: An Overview of Data Science Versioning Practices and Methods. In: 16th SC@RUG 2019 Proceedings 2018-2019. Bibliotheek der R.U., 2019.

[Sc21] Schüle, M. E. et al.: TardisDB: Extending SQL to Support Versioning. In: Proceedings of the 2021 International Conference on Management of Data. ACM, pp. 2775–2778, isbn: 978-1-4503-8343-1, doi: 10.1145/3448016.3452767, 2021.

[Se12] Seering, A. et al.: Efficient Versioning for Scientific Array Databases. In: 2012 IEEE 28th International Conference on Data Engineering. IEEE, pp. 1013–1024, isbn: 978-0-7695-4747-3, doi: 10.1109/ICDE.2012.102, 2012.

[Ta15] Taylor, K. et al.: A Provenance Maturity Model. In: Environmental Software Systems. Infrastructures, Services and Applications. Vol. 448, Springer International Publishing, pp. 1–18, isbn: 978-3-319-15993-5, doi: 10.1007/978-3-319-15994-2_1, 2015.

[Ti85] Tichy, W. F.: RCS — a System for Version Control. Software: Practice and Experience 15(7), pp. 637–654, issn: 0038-0644, 1097-024X, doi: 10.1002/spe.4380150703, 1985.