

DSDB: Reproducible Computational Modeling

Jackson Brown, Nicholas Weber
Allen Institute for Cell Science, University of Washington

Abstract—In the following abstract we describe Dataset-Database (DSDB), an open-source system for handling the provenance, versioning, de-duplication, history, and query of dynamic databases used at the Allen Institute for Cell Science. We present our initial design and deployment of DSDB, the results of this work for computational modeling, and conclude with a discussion of the future work necessary for provisioning data discovery and sharing tools that facilitate transparent reproducible research through provenance aware features.

I. INTRODUCTION

This abstract describes a set of tools designed to meet the version control, de-duplication, and data packaging needs of computational modeling researchers at the Allen Institute for Cell Science. Computational modeling at the Allen Institute commonly produce multiple and intermediate versions of datasets, including analysis results, model predictions, training weights and parameters for neural networks [1]. In many cases, there is no clearly defined “final” version of a dataset which then becomes canonical to “final” results. As research is conducted, algorithms are tested, results are generated, and a network of implicit dependencies is created within a single database. While dataset context can be provided when using other versioning systems (e.g. commit messages detailing how files have changed) these tools are not fully automated, and other provenance tracking systems do not satisfactorily capture the environmental dependencies used to change data that are stored in traditional relational database models [2]. To natively provide the tooling necessary to encapsulate a modeling environment, or, at minimum, to rebuild the environment, new approaches to data provenance are needed.

To address challenges in data storage and provenance at the Allen Institute for Cell Science, we created a relational database schema and developed a supporting Python library, DatasetDatabase (DSDB). DSDB facilitates sharing, dataset deduplication, and processing of data that integrates with modeling workflows, and, by extension, common computational modeling workflows. DSDB attempts to solve the impediments to context tracking inherent in previous versioning systems (e.g. by enforcing immutable datasets - dataset changes only occur during the runtime of approved system functions) [3]. This makes it incredibly easy for DSDB to encapsulate an analytic environment, and more importantly, makes it simple for the user to record and communicate provenance information in the database itself.

II. USE CASE

The challenges that DSDB solves might best be described through a user story that is typical of the work conducted by a computational scientist at the Allen Institute for Cell Science:

Table I
EXAMPLE MITOTIC STAGE TRAINING DATASET

CellId	Filepath	Rev. 1	Rev. 2
1	/projects/...	0	0
2	/projects/...	4	3
3	/projects/...	0	0
4	/projects/...	1	1

A researcher on the modeling team needs to create an imaging-based machine learning model that takes in images of single cells and outputs a mitotic stage classification for each cell. Table 1 is an example of such a training dataset.

In this example, each cell has been assigned a unique cell id and is accompanied by filepath to an image of the cell and two manually classified mitotic stages provided by scientists from the assay development team at the institute.

A scientist from the assay development team will typically store this dataset as a comma-separated-values file, or ‘CSV’. The delivery of this dataset is done by saving the dataset as a CSV to a shared network storage drive. The creator(s) of the dataset will then send a message, by email or messaging service, to the primary researcher(s) for the project on the modeling team that includes details regarding the dataset’s creation and includes the network storage drive filepath to the dataset. Modeling team members will then make their own copy of the dataset. The two most common reasons for duplicating the dataset at this stage are for faster file read times in comparison to reading the file from the shared network drive, or to move it to a location in their own project directory so as to have it referenced locally within the project.

This process creates multiple copies of a dataset that need to be managed and identified for reproducibility. This is problematic because a modeling team member will often identify errors or unstructured portions of the data. The most common errors for imaging based datasets are invalid filepaths, or encountering a file that was once present but was later accidentally moved or deleted by the creator.

A. Unstructured Data

Another common error, which can be classified as an unstructured data error, is when too many values, too few values, or varied data formats are provided. An example of this can be seen in Table 2.

Using Table 2 as an example, the expected structure for both reviewer’s mitotic stage classifications is a single integer value between zero and seven; but for cell id 21, reviewer one has recorded what can be interpreted as an uncertainty regarding the mitotic stage because they have left two integers separated by a

Table II
EXAMPLE OF UNSTRUCTURED DATA ERRORS

CellId	Filepath	Rev. 1	Rev. 2
21	/projects/...	3,4	3
22	/projects/...	0	
23	/projects/...	five	5

The expected structure of both the “Rev. 1” and “Rev. 2” is to have a single integer value between zero and seven.

comma. Separately, for cell id 22, reviewer two has left their classification for mitotic stage blank. More uncommon, but still present, is the example of cell id 23, when a completely different data format was used; in this case, text of the word ‘five’ was used by reviewer one instead of the integer five (5). When errors in the dataset are discovered, they are reported back to the team that created the dataset.

B. Additional Data Requests

An additional common hindrance in data exchange is when a modeling team member needs more data delivered than was originally requested. In essence this is a schema change to the dataset (i.e. a research team member would write that “we need a column detailing additional features of the cell in question.”) This exchange process will happen successive times between team members; a dataset is handed from the data producer to the modeling team, checked for errors, inconsistencies, or increased data requirements, and the modeling team will request resolutions to issues or more data from the data producer. Problematically, model prototyping often begins before all final data error resolutions and schema change requests are made between teams. As a result, multiple versions of the same dataset may be present in a project and each version may only be very minimally different from a prior version. Although each dataset may have distinct versions, the identification and resolution of these errors becomes difficult to manage.

C. Tracking Computational Provenance

Lastly, the problem of providing data provenance to the intermediate datasets that are created during modeling computation. Modeling team members will generally store these intermediate datasets as additional CSVs in their project directory. These intermediate datasets are usually given limited details and metadata regarding the scientist who initiated the computation and what environment or algorithm was used for creation. There are other minor changes made continually to the base dataset, however the previous examples demonstrate a majority of computational modeling data delivery mechanisms used throughout the data handoff process.

III. PROOF OF CONCEPT - DATA STORAGE

As described in the user stories above, the research teams at the Allen Institute for Cell Science require a system that can work as a shared data storage solution (like that of a system for version control of databases like OrpheusDB), but with more flexibility in schema, minimal impact on workflow, and natural enforcement of dataset context and provenance (like that of

a provenance workflow tool native to Python, like ReciPy). The simple combination of these two systems is limited in two ways: first is the rigid dataset schema issue introduced by OrpheusDB, and the second is how to manage filepaths, or files, that are commonly referenced within datasets.

OrpheusDB operates by adding a versioning table that references a foreign key from the data table that is being versioned, and with this key a user can create a series of versioned data tables by simply recording a collection of foreign keys, or which rows of the data table, as a version of the linked data table. In this sense, OrpheusDB efficiently handles granular data level changes (e.g. values in a column that need to be updated or added). However, OrpheusDB’s solution to versioning is not suitable for dataset schema changes (i.e. an entire column needs to be added or removed from the table (dataset)). To properly handle versioning dataset schema changes under an OrpheusDB model, you would need to create a new table every time a dataset schema change occurs.

Dataset schema changes occur frequently on computational modeling datasets, requiring a different approach to versioning. DatasetDatabase (DSDB) solves this issue by taking the concepts OrpheusDB has established for granular data versioning and expanding on them an additional step. Instead of only recording which rows are contained within each version of a table, DSDB additionally records which values are contained within each row. This is known as “Dataset - Group - Iota” deconstruction.

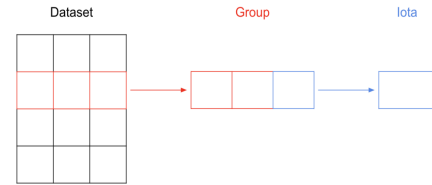


Figure 1. Dataset - Group - Iota Deconstruction

As seen in Figure 1, Datasets may consist of multiple Groups, and Groups may consist of multiple Iota. DSDB accomplishes this by doing an extensive dataset deconstruction process¹. When a user attempts to upload or ingest a dataset into the DSDB system, the deconstruction process begins by tearing the dataset into rows (Groups) and further strips rows (Groups) into single cells (Iota). Iota ingestion occurs first and Iota are stored discretely. The deconstruction of Table 1 would result in Table 3.

The next step is to create a unique Group for the ordered list of Iota that make up the row. This is done by generating a SHA256 hash value for the ordered list of IotaIds that made up the row. Completing this operation using Table 1 and 3 would result in Table 4.

¹As a note, this process could have been called the “Dataset - Row - Cell” deconstruction, however Group and Iota were chosen to reduce terminology overload, as “cell”, is also a common term in biological research.

Table III
IOTA TABLE

IotaId	Key	Value	Created
1	CellId	b*\x80\x0...	2019-01-...
2	Filepath	b*\x80\x0...	2019-01-...
3	Rev. 1	b*\x80\x0...	2019-01-...
4	Rev. 2	b*\x80\x0...	2019-01-...
5	CellId	b*\x80\x0...	2019-01-...
6	Filepath	b*\x80\x0...	2019-01-...
7	Rev. 1	b*\x80\x0...	2019-01-...
8	Rev. 2	b*\x80\x0...	2019-01-...
...

Example of Iota created from the first two rows of Table 1. Key, is a reference to the cell's column name. Value, is the binary dump of the cell value.

Table IV
GROUP TABLE

GroupId	SHA256	Created
1	3dcba0549b496d23714...	2019-01-...
2	3799273ecc32e328403...	2019-01-...
3	5899abb43952eff27ba...	2019-01-...
4	75d34df97c51bcf2471...	2019-01-...

Example of Groups created from deconstructing Table 1. SHA256 is the hash of the ordered list of Iota ids that make up the Group. Ex: GroupId 1 in this table corresponds to the SHA256 hash of [1, 2, 3, 4], the Iota ids in Table 3 created from row 1 of Table 1.

It is worth noting that in Table 4 there is no metadata connecting the created Groups to any specific Dataset - only a unique ordered list of Iota was registered during ingestion. Connection information (metadata) is stored in join or junction tables. Iota and Group tables have a join table that links an IotaId to a GroupId, and Group and Dataset have a join table that link a GroupId to a DatasetId. The Iota-Group join table does not have any extra information other than the foreign key relation between the two tables, however, the Group - Dataset join table stores an additional label attribute, which is most commonly the row index to place the linked group at in the Pandas dataframe during dataset reconstruction. With this schema, DSDB is able to minimize redundant data storage; only unique Iota and Groups may ever exist. Rather, it is the connections between these pieces that constructs a dataset.

The Dataset table stores metadata about each dataset added by users of the system. This includes items such as name, description, created datetime, and most importantly, a SHA256 hash for the dataset. When a user attempts to upload a dataset to the system, before any database ingestion is run, the supporting Python library generates a SHA256 hash for the dataset using a list of transactionally generated Group ids ordered by their accompanying label (row index). If the generated hash is found in the Dataset table, DSDB will rollback the transaction and point the user at the already stored dataset. This is both a validation and enforcement mechanism created by the schema of DSDB. By enforcing only unique Iota, Groups, and Datasets can be entered into the database, users are quickly made aware when other users have already uploaded the same dataset.

A. Deduplication & Validation

One of the major goals of DSDB's schema design was to minimize redundant data storage. At all levels, data is attempted to be deduplicated. Only unique Iota can exist in the Iota table (the unique set of key, and binary value); only unique Groups can exist in the Group table (the unique SHA256 hash of the ordered list of Iota ids); only unique Datasets can exist in the Dataset table (the unique SHA256 of the ordered list of Group ids).

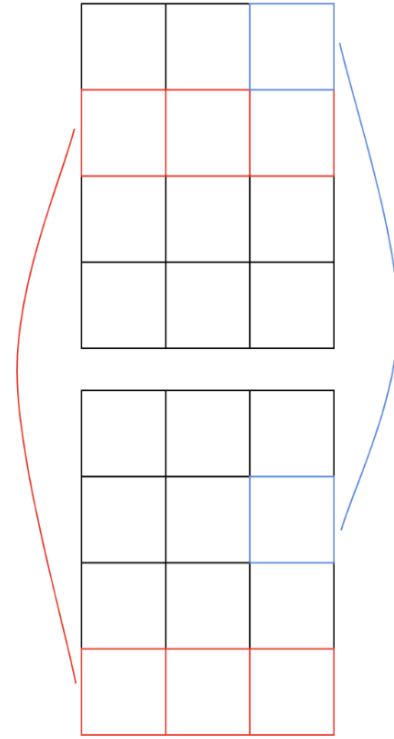


Figure 2. Cross Dataset Iota and Group Deduplication

The Dataset - Group - Iota deconstruction process additionally allows for the validation of every Iota value before ingestion. The supporting tooling allows the user to validate every Iota, both for schema and data validity. Columns (Iota keys) marked as filepaths can additionally be verified for existence, and the associated file can be stored using a unique file storage system. For the Allen Institute, this represents a centralized file storage server that provides a globally unique identifier (GUID) as well as an immutable filepath for a copy of any file entered into the system.

Unstructured data errors, as in the example of Table 2, can also be handled during the Iota validation process. If a user attempts to upload a dataset, and specifies that 'Rev. 1' column values should be integers, but the string "3,4", list [3, 4], or string 'three' was retrieved, an error would be raised. The database schema of DSDB naturally encourages dataset cleaning and dataset schema enforcement; however, as these

Table V
ESTIMATED DATASET RETRIEVAL TIMES ON ALLEN INSTITUTE
NETWORK

Dataset Size	Local (seconds)	Cluster (seconds)
10,000	8.74	8.64
100,000	15.80	12.18
1,000,000	93.58	48.03
10,000,000	1592.59	450.53

Estimated retrieval times for local and cluster machines using quadratic fits. Dataset size is equal to rows \times columns.

cleaning and validation checks are provided by supporting tooling - external to DSDB - their use is not required.

B. Retrieval

To retrieve a dataset from the database, the user is provided a function by the DSDB library that performs all the table joins, and reforms Iota and Groups back into a Pandas dataframe, given a dataset id. Because modeling team members are consistently accessing and uploading data, retrieval times are an important benchmark for DSDB. To better record and document how the Allen Institute implementation of DSDB was scaling, a DSDB report generation tool was created to provide details about both database size but most important to users, dataset retrieval times.

Figure 3. Dataset Retrieval Times against Dataset Size

Figure 3 shows dataset retrieval times in seconds against dataset size (row \times columns) tested against two machines, a laptop used as a local machine (CPU MHz: 2500, Physical Cores: 2, Logical Cores: 4), while the higher end (CPU MHz: 3000, Physical Cores: 16, Logical Cores: 32) is a node on the institute shared computation cluster. What this figure does not address is the network speed difference from being on a local machine to being on the shared cluster. The only time this matters is when transferring the actual data from the database to the requesting machine. The bulk of the workload cost of DSDB comes from the reading of Iota values back into memory and formatting the Iota and Groups back into a Pandas dataframe. (For specific retrieval time estimates see Table 5.)

A final statistic created by the report generation tool was that of Iota deduplication, computed by dividing the length of the Iota table by the sum of all dataset sizes (row \times column), or, how many Iota exist in the database over how many could exist without deduplication efforts. The Allen Institute implementation of DSDB currently reports 49% Iota deduplication. This is expected as many of the projects and datasets currently using DSDB are classification model training datasets, which have similar dataset schemas and a limited range in data values.

DSDB’s storage solution solves many of the modeling team’s management and retrieval problems, but DSDB also provides relevant query and discovery capabilities. Under DSDB, there is a centralized table of rich metadata about any

unique dataset that has ever been uploaded as well as more complex, but even more rich, query capabilities available to users which require more intensive searches. If a researchers asks, “Which datasets has this image (filepath) been used in previously” DSDB can answer this query quickly under the database schema as a simple operation of querying the Iota table’s value column with the binary dump of the filepath in question. In the case where the file in question was sent to a custom file storage service prior to dataset ingestion, the original filepath would first need to be looked up from that service then the standard query could be completed.

Basic text-based dataset discovery is implemented by providing a query function in the supporting library that uses the sum of computed term frequency - inverse document frequency (TF-IDF) scores for the terms used in each dataset description against a search phrase or term sequence.

A more complex query that a user could complete could take into account percent mutual information between datasets. As it is common that users may forget to attach dataset metadata during upload, a dataset may have limited or no text based metadata to query against. A naive implementation of this functionality would involve computing a TF-IDF score for every dataset, then return the maximum value found from multiplying the computed TF-IDF score for each dataset by a percentage shared Iota between every other dataset.

A database schema diagram for DSDB available at Appendix A.

IV. LIMITATIONS

Although we have demonstrated a number of efficiency gains in storing, versioning, and documenting data through DSDB, there are a number of known limitations to the current release. Storing and versioning large datasets has slowed retrieval time down significantly. For Allen Institute team members, slow retrieval is not an issue when running multiple hour long feature extraction or training applications as there is no need for a user to view the intermediate datasets as they are created. However, delayed retrieval will be an issue when a user simply wants to view a large dataset. As was demonstrated in Table 5, datasets with less than 1,000,000 cells can be retrieved in less than a minute (48 seconds) on the shared computation cluster. Scaling the performance of DSDB with datasets larger than 1,000,000 cells is a future research challenge to be addressed in version two.

A similar limitation is found when datasets that are stored in DSDB are not created from the DSDB apply function, but rather simply uploaded. If a majority of datasets are being uploaded rather than created while using the provenance features, the issue of initial dataset retrieval times is exaggerated as this is the only functionality many users will interact with; users are only seeing one set of beneficial features instead of the entire suite of features available.

V. FUTURE WORK

We have documented the initial motivation for, and design of DatasetDatabase (DSDB) for handling the provenance,

versioning, de-duplication, history, and query of dynamic databases in order to enable verifiable and shareable research results. The main use cases satisfied by DSDB are from scientists at the Allen Institute for Cell Science - where DSDB is currently a production system for scientists cleaning and preparing imaging and cell feature datasets. Of the many cleaning and validation features in DSDB, dataset uniqueness, or the ability to determine which datasets have entered the system previously, has been the most popular for capturing and communicating provenance across teams. The immutability constraints that DSDB imposes also allows collaborators on projects to easily verify which dataset to use for early prototype work by simply sharing a dataset ID instead of a filepath that could be easily changed or moved.

Given the limitations of DSDB 1.0 we believe the important future developments should focus on: 1. Provenance logging that is similar to ReciPy. The addition of moving generated files and datasets to a shared stored instead of simply being referenced in ReciPy logs would increase shared dataset provenance; or, 2. Move DSDB from a relational database to a graph database. While there are tradeoffs to this migration, a number of previous studies show how queries that are table-join heavy under relational databases become simple under graph databases [4]. Both options are primarily concerned with retrieval times, the main difference being: storing files (more ReciPy oriented), or continuing with the “Dataset - Group - Iota” deconstruction (moving to a graph database).

VI. ACKNOWLEDGEMENTS

We wish to thank the Allen Institute for Cell Science founder, Paul G. Allen, for his vision, encouragement, and support. This work could not have been completed without the additional support and input from all members of the Allen Institute for Cell Science modeling team.

APPENDIX

APPENDIX A

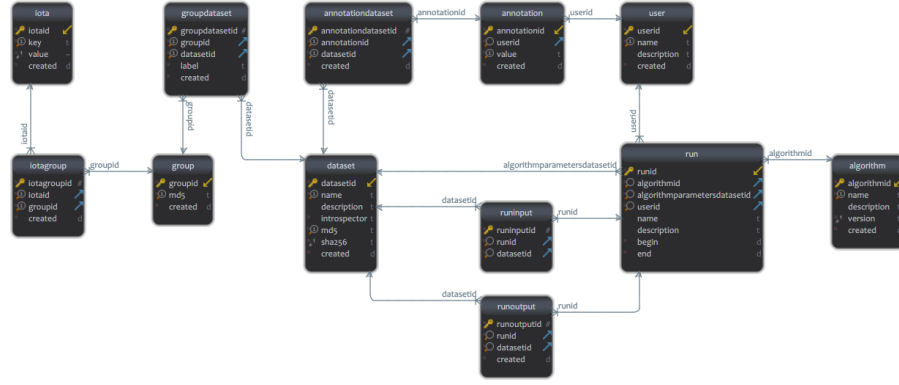


Figure 4. Version 1.2 DSDB Database Schema

REFERENCES

- [1] D. Yuan, Y. Yang, X. Liu, and J. Chen, "On-demand minimum cost benchmarking for intermediate dataset storage in scientific cloud workflow systems," *Journal of Parallel and Distributed Computing*, vol. 71, no. 2, pp. 316 – 332, 2011, data Intensive Computing. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0743731510001838>
- [2] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, "A comparison of a graph database and a relational database: a data provenance perspective," in *Proceedings of the 48th annual Southeast regional conference*. ACM, 2010, p. 42.
- [3] M. Imran, H. Hlavacs, I. U. Haq, B. Jan, F. A. Khan, and A. Ahmad, "Provenance based data integrity checking and verification in cloud environments," *PloS one*, vol. 12, no. 5, p. e0177576, 2017.
- [4] R. Angles, "A Comparison of Current Graph Database Models," 04 2012, pp. 171–177.