DBgen: A Python Library for Defining Scalable, Maintainable, Accessible, Reconfigurable, Transparent (SMART) Data Pipelines

Michael J. Statt[1], Kristopher S. Brown[1], Santosh Suram[2], Linda Hung[2], Daniel Schweigert[2], John Gregoire[3], Brian A. Rohr[1,*]

Abstract

In this work, we present DBgen, a Python library that provides a framework for defining extract-transform-load (ETL) pipelines to create and populate SQL databases. DBgen is most useful when the underlying data has complex relationships, requires multi-step analysis, is large-scale, and the type of data being collected changes frequently. Scientific data often fits this description. With current tooling, defining ETL pipelines for this particularly difficult-to-manage data is so onerous that a great deal of it does not end up being stored in a database and is opaque. DBgen is designed to fill the gap in the current tooling and reduce the barrier to defining ETL pipelines such data.

Keywords: Database, ETL, Python, Data management

*Correspondance to:
Email address: brian.rohr@modelyst.io (Brian A. Rohr)
[1]Modelyst LLC, Palo Alto, CA 94306, USA
[2]Toyota Research Institute, Los Altos, CA 94022, USA
[3]California Institute of Technology, Pasadena, CA, USA

Required Metadata

Current code version

Nr.	Code metadata description	Please fill in this column
C1	Current code version	1.0.0
C2	Permanent link to code/repository used for this code version	*https://github.com/modelyst/dbgen*
C4	Legal Code License	Apache 2.0
C5	Code versioning system used	git
C6	Software code languages, tools, and services used	Python 3 on Linux, OSX or Windows. Dependencies listed in requirements.txt in code repository.
C7	Compilation requirements, operating environments & dependencies	N/A
C8	Link to developer documentation/manual	https://www.dbgen.modelyst.com
C9	Support email for questions	brian.rohr@modelyst.io

Table 1: Code metadata

1. Motivation and Significance

DBgen is designed to reduce the barrier to creating databases for complicated data sources in accordance with the FAIR data principle[1]. The FAIR principle, which states that data should be Findable, Accessible, Interoperable, and Reusable, is widely accepted and has shown utility in a variety of fields of scientific research including medicine[2, 3, 4], meteorology & oceanography[5, 6, 7, 8], oral speech studies[9], botany[10], mass spectrometry[11], and many others. Although many agree that it is important to make data FAIR, a great deal of scientific data remains stored in a way that does not meet these principles. We contend that this is in part because defining extract-transform-load (ETL) pipelines to get scientific data into SQL databases is a particularly challenging task. Scientific data is complex, analysis-heavy, frequently-changing, large-scale, and the people who understand this particularly difficult-to-manage data best are rarely experts in SQL and data engineering. These challenges regarding making scientific data fair have been highlighted previously [12, 13, 14, 15, 16] as has the importance of directing resources toward solving these infrastructure problems [17]. In order to handle this challenging case, scientific ETL pipelines need to be scalable, maintainable, accessible, reconfigurable, and transparent

(SMART). In section 1.1, we describe how the SMART principles address the challenges in defining scientific ETL pipelines. Then, in section 1.2, we identify a gap in the current tooling which makes it difficult to define SMART ETL pipelines. In section 2, we describe what DBgen is and how it helps to fill this gap in the current tooling.

1.1. SMART ETL Pipelines

Scalable: As high-throughput experimentation becomes more widely used, the scale of scientific data continues to increase. This demands that any scientific ETL pipeline can handle large amounts of data efficiently.

Maintainable: The fundamental relationships between the entities of interest in scientific data are very complex. There are many of types of entities that need to be tracked, and the relations between these entities are oftentimes many-to-many. There is also a great deal of meta-data that needs to be stored and formally linked to the data. This makes the FAIR principles of rich meta-data and strong provenance challenging to achieve. A database architecture capable of capturing this complexity requires many tables and foreign keys. Accordingly, the ETL pipeline that populates such a database is a fairly complex piece of software. As with any complex piece of software, maintainability is crucial, and making the code modular and easy to debug is imperative for maintainability.

Accessible: The number of scientific researchers who are comfortable with higher-level languages like Python far exceeds the number who are comfortable with SQL, so in order to make ETL pipelines editable by scientists, it is important to abstract away the SQL code.

Reconfigurable: Scientific ETL pipelines must be easily reconfigurable because scientific research is inherently frequently-changing. Scientists frequently decide to do new types of experiments and analyses as the very goal of doing the research is to learn information that changes their understanding of the subject of research. As they do new types of experiments, they start to track new types of information and carry out new analyses. Changing the entities, attributes, and relations that are tracked in a database is the definition of a schema change. Therefore, schema changes occur far more often in scientific research than in other fields, and the ETL pipeline needs to be easy to reconfigure when these inevitable changes occur.

Transparent: In many cases, researchers are interested in a high-level analysis of the data, not just the raw data itself. The process of converting raw data to high-level results usually requires many small steps. Each data processing step yields an intermediate result, which may be of interest in its own right or may be useful in other analyses later. When the data pipeline is complete, there can be a large, complicated web of data processing steps between the raw data and the final results. It is imperative that the full flow of data from the original, raw, file all the way through to the final analysis is transparent.

1.2. Current Tooling

There are many tools, including DBeaver, TablePlus, and MySQL Workbench, that make it easy to define complex empty database schemas; however, in the case of scientific data, defining the ETL pipeline is much more difficult than defining the empty database schema. There are several existing Python packages, including psycopg2, pymysql, and sqlalchemy that expose SQL functionality to a python user, but they do not provide any framework for defining SMART ETL pipelines. DBgen fills this gap in the current tooling. If one were to implement 50 complex, SMART ETL pipelines, each for a different use case, yes, some of the code would specific to each use case, but a large amount of the code would be common to all of the use cases. DBgen is that code that is common code. In other words, DBgen is a Python package that provides a framework for the definition of SMART ETL pipelines, just as pytorch [18] and tensorflow [19] provide a framework for the definition of GPU-accelerated, complex deep learning models.

2. Software Description

2.1. Software Architecture

Each DBgen Model defines a complete build procedure for a database: both instantiating the empty database and populating it with data. The DBgen Model consists of two graphs, one for each of those steps. The schema graph defines the empty database architecture, and the ETL graph defines the ETL process, which populates the database with data.

In DBgen, there are three key classes associated with defining the schema graph (Entities, Attributes, and Relations), and four key classes associated with defining the ETL graph (Generators, Queries, PyBlocks, and Loads). Both pieces are put together in one, large object, called a Model, which defines the entire database build procedure, both instantiating the database and populating it.

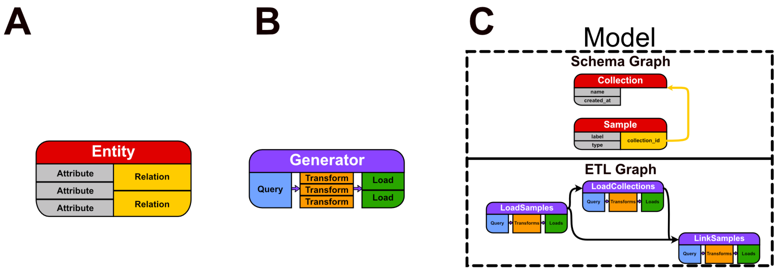

Figure 1: A depiction of the relationships between the key classes in DBgen. (A) shows how the classes that comprise the schema graph relate to each other. (B) shows how the classes that comprise the ETL graph relate to each other. (C) shows how many instances of these classes can be composed to create a DBgen Model.

2.2. The Schema Graph: Defining the Empty Database Schema

Entity: In DBgen, each Entity fully defines an empty database table. It consists of a name and any number of Attributes and Relations, which are described next. Each entity is a node in the schema graph.

Attributes: Attributes, define the columns of each database table. They have a name, a description, and a data type. Attributes in DBgen can be either "identifying" or "non-identifying." The information stored in the identifying columns are necessary and sufficient to identify exactly one row in a given table. DBgen disallows the existence of two rows in the same table with the same identifying data. For example, in a table of movies, one could decide to make the title and release date Attributes identifying. In DBgen, this would guarantee that a query for a specific title and release date would return no more than one row, and DBgen would also require a title and a release date to create a row in the table. There may be many other non-identifying Attributes, like duration and average critic rating. Although the back-end details are different, this concept is analogous to a composite primary key.

Relations: Relations define the relationships between the tables and therefore the edges of the schema graph. For those who are familiar with database terminology, a Relation defines a foreign key.

2.3. Populating the Database

The ETL graph consists of Generators, which are the nodes in the ETL graph. The information specified in the Query and Loads objects, also described below, allow DBgen to automatically compute the appropriate edges for the ETL graph.

Generators: Each Generator, defines a single extract-transform-load (ETL) step. It consists of a Query, a PyBlock, and Loads, which represent the extract, transform, and load steps, respectively. A common pattern in DBgen data pipelines is to query the database, use that information as inputs to a function, and insert the result back into the database. This allows for complicated, multi-step data analyses to be carried out in a flexible, modular, maintainable, and transparent manner.

Query: The Query object in DBgen defines a SQL query. By using a Query object rather than a raw SQL string, the Generator knows which database columns need to be queried in order for that ETL step to run. DBgen will later use this information to compute dependencies among the ETL steps and automatically run them in the correct order.

PyBlock: The PyBlock object represents the transform portion of the ETL step. It is any arbitrary python function, which enables sophisticated analyses, including predictions from machine learning models, to be run. The inputs to the function come from the Query object, and the outputs from the function are inserted back into the database in the load step, which is described next. Pyblocks can also read data in from the file system, which is commonly used for the early ETL steps in the data pipeline.

Loads: As the name suggests, Loads represent the load portion of a given ETL step, which is the step in which data is inserted into the database. In an instance of the Loads object, the user specifies which outputs from the PyBlock are inserted into which columns in the database. Now, each Gen knows which database columns must be populated before it is run (from the Query object) and also which database columns are populated by that Gen (from the Loads object). DBgen needs this information to determine the correct order to run the ETL steps.

Model: Finally, the Model object defines the entire database build procedure. It is comprised of a set of Entities and a set of Generators. When a Model is run, it uses the information in the Entities to create the schema graph and to instantiate the empty database tables. Then, it uses the information from each Generator's Query and Load objects to compute the order in which the Gens need to be run, thereby creating the ETL graph. Finally, it executes each Generator's Query, Pyblock, and Loads steps to actually populate the database.

2.4. Software Functionalities

Automatic Ordering: Complicated data pipelines are comprised of a large number of small, simple processing steps. Without DBgen, the user must make sure that the ETL steps are run in the correct order. If one ETL step needs to query a given column in the database, the ETL step that populates that column needs to be run first. This process is laborious, especially in the case of scientific research, where frequent changes to the data pipeline are expected. DBgen automatically determines the correct order to run the ETL steps by using the structured information in each Generator object to create a directed acyclic graph (DAG). This feature is significant because it allows the user to not think about the whole data pipeline when adding or editing an ETL step, even if the change completely disrupts the dependencies among the ETL steps. This feature contributes to the reconfigurability and maintainability of DBgen ETL pipelines. Furthermore, this computational graph can be visualized to show the full flow of data from its source, through all processing steps, to the final destination in the database. This adds transparency to all DBgen ETL pipelines.

Primary Key and Foreign Key Handling: In large, complex database schemas with many foreign keys, querying tables to populate these foreign keys properly is both computationally expensive and laborious for the user to write. DBgen obviates the need for this altogether. The technical details of how DBgen accomplishes this are described in the supplemental information.

Automatic Detection of Changing Inputs and Functions: Every time a DBgen ETL step is run, DBgen stores a hash of the inputs it received and the function that processed the data. Then, when a pipeline is re-run, DBgen automatically knows which steps need to be re-run. This avoids the computational expense of re-running functions, and perhaps more importantly, it allows the user to add data and make edits to the pipeline without thinking about the execution of the pipeline at all. If the user wants to change a function, they just change the function and re-run the DBgen model. DBgen will automatically detect that only that function was changed, and that ETL step and all of its dependents will be re-run, and the rest of the steps will be skipped, as their results are unchanged. This is a recurring theme in DBgen: separate the definition of the pipeline from the execution of the pipeline, and abstract away the execution portion, thereby enabling the user to zoom in and make edits on any small portion of the pipeline without needing to think at all about how that may impact the broader data pipeline. This contributes to the reconfigurability and scalability of DBgen ETL pipelines.

DBgen Log Database - A Dashboard for the ETL Process: The log database is a separate, small database that is designed to be a dashboard for the data pipeline that helps the user debug complicated data pipelines with ease. The log database has one row per ETL step per attempt at running the data pipeline. The most common use case is to query this table for the most recent run attempt. For each row, it contains the following information:

- The status of the ETL step (not started, failed, completed, etc.)

- The runtime for the ETL step
- The full traceback for any errors that were encountered when trying to run the ETL step
- Which columns the ETL step queries
- Which columns the ETL step populates

So, if there is an error in one of the functions, and a user tries to run the data pipeline, the user can go to the dashboard and immediately see: these steps finished, this step encountered an error, all of its dependencies did not run as a result, and here is the error message. This makes even the most complicated ETL pipelines easy to debug and maintain. Even when there are no errors, the user can query the runtime of each step to identify bottlenecks in the process. The DBgen log database makes

Efficient, Cloud-Optimized Insertion Methods: DBgen copies data from the machine that is executing the ETL pipeline to the machine that is hosting the database in large batches, which is more computationally efficient than traditional methods. Additionally, DBgen makes it easy to run each individual ETL step on a separate compute instance, so if one ETL step is particularly compute-intensive, it can be run on a more powerful compute instance. Together, these features ensure that all DBgen ETL pipelines are scalable. The details are described in the supplemental information.

Abstraction: With DBgen, a user can implement an entire database and ETL pipeline without writing one line of SQL code. This provides accessibility to ETL pipelines defined using DBgen. With the SQL abstracted away, a much broader user base is able to define or at least understand and edit existing ETL pipelines.

3. Illustrative Example

The example we will discuss is a database of materials science research data created and populated using DBgen called ESAMP, which stands for "Event-Sourced Architecture for Materials Provenance." To give a sense of the inherent complexity of the problem, in materials science research, a material sample of interest is usually derived from one or many other samples through a series of procedures. Each procedure may produce one or many data files, which may be used in one or many analyses to yield results that are of interest to the researcher. Some analyses depend on other analyses having already been carried out. A query against the database should be able to answer complicated questions like, "show the top 10 best-performing batteries as determined by a specific lifetime test that were derived from anodes that are at least 10% cobalt as determined by X-ray photo-electron spectroscopy (XPS), and exclude any batteries that have the solvent that arrived on May 18th anywhere in their processing history."

ESAMP models this complexity completely without making any simplifying assumptions. This is significant because it adds transparency, flexibility, and provenance to the curation of datasets for machine learning (ML) or other data analysis techniques. When the underlying data is modeled without assumptions, the user is able to write a SQL query with a certain set of constraints and assumptions to generate a dataset. If the researcher finds that the data is imbalanced or comes across another problem with their first dataset, they can easily edit the query to generate a new, improved dataset.

Importantly, any future researcher who looks at that dataset and wants to know exactly where the data came from and what assumptions and constraints were applied can simply look at the SQL query that was used to generate it. This provides much-needed transparency in materials science ML projects.

The underlying data that is now in the ESAMP database was originally stored in a large zip archive containing additional zip archives containing tens of thousands of automatically-generated, custom-structured text files. The data pipeline that extracts the data from that structure and populates the ESAMP database requires over 50 ETL steps with a complicated tree of dependencies. The process of designing the ESAMP architecture required a great deal of iteration, so the ETL pipeline had to be adjusted and redefined dozens of times. Accomplishing this without DBgen would have been prohibitively laborious.

3.0.1. Generator Example

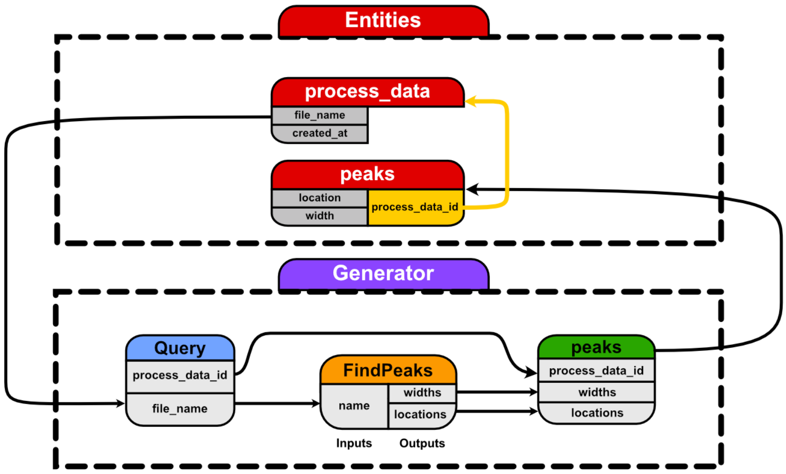

Figure 2: An example of the flow of information within a Generator

Figure 2 illustrates the flow of information within an example Generator. In this example, process data has been stored within the process_data Entity using its file_name and the goal of this Generator is to extract each file and find the locations and widths of the peaks within the underlying data. As this generator queries the file_name from the process_data Entity and loads the resulting peaks into the peaks entity, DBgen automatically places this Generator after any Generator that populates the process_data entity and before any Generator that depends on the peaks Entity being populated. Therefore, the author of this Generator can narrow their field of vision to just the data flowing from the data extraction to the loading. As discussed above, DBgen is specifically designed to allow for the authors of the pipeline to easily and effectively change their mind. The three common types of changes made to a pipeline are:

1. New data entering the pipeline
2. New functions to process the data
3. New schema for storing the inputs and outputs

The severity of these changes typically increases from 1 to 3, with an updated schema being the most extreme. However, DBgen helps to greatly reduce the complexity of implementing each type of change.

Firstly, detecting whether a file entering the pipeline has been seen before is critical to reducing the computational strain on the overall pipeline. To avoid reprocessing of duplicate data, this generator will store a hash of the generator and each input in the DBgen Log Database discussed above. This enables DBgen to automatically detect whether it has seen an extracted input before and, if so, it can skip reprocessing the duplicate row. While this may seem trivial for a path to a file, this duplicate detection generalizes to complex inputs that are aggregated and processed from many entities within the schema.

Secondly, scientists regularly update and improve the functions they use to transform data. This could be a change to the FindPeaks transform or the addition of a pre-processing step to remove the outliers in the data before the FindPeaks transform processes the data (as shown in Figure 3). In either case, DBgen makes these changes easy to make by automatically ordering the functions within a generator. This means that FindPeaks need only request an output from the RemoveOutlier for DBgen to know that RemoveOutlier needs to be run prior to FindPeaks. Additionally, the generator's hash will change with the addition of a transform or the modification of any transforms underlying code. This signals to DBgen that each input must be reprocessed regardless of whether it had been processed by the previous version of the generator.

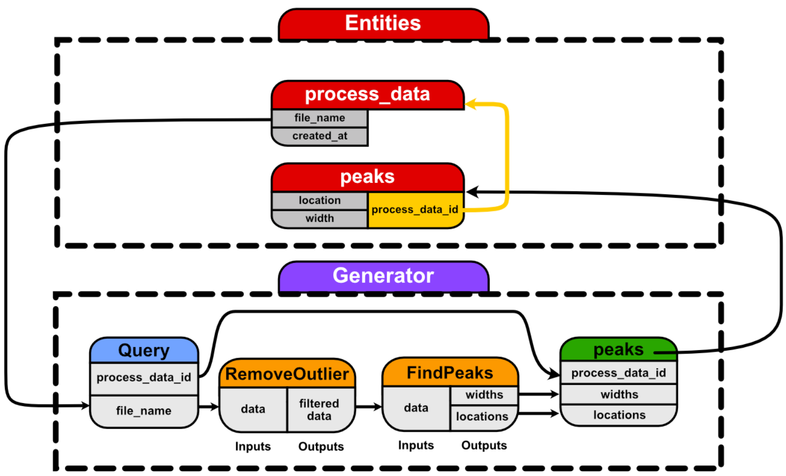

Figure 3: A generator with two PyBlocks, or transform steps.

Finally, schema changes can be facilitated by DBgen by isolating the schema that stores the data from the generator that modifies it through the use clear, well-defined interfaces of Loads and Extracts. When schema changes are made, such as the modification of the relationship between process_data and peaks from a one-to-many to a many-to-many relationship, each generator that depends upon or populates the modified entities in the schema is clearly logged in the DBgen Log Database. Additionally, the abstractions of Query objects and Load objects allows for complex schema changes to be accommodated by only a few lines of code.

4. Impact

DBgen provides a framework that makes it easy to implement SMART ETL pipelines. We believe this will have the largest impact in the field of scientific research. Specifically, we believe that DBgen will result in more scientific data being stored in accordance with the FAIR principles of data management. Ultimately, this will accelerate innovation in computational methods applied to experimental scientific data. As a point of analogy, ImageNet [20] was published in 2009 and played a significant role in the development of new convolutional neural network architectures shortly thereafter, as evidenced by the publication of AlexNet in 2012[21], and Resnet in 2016[22]. In many fields of scientific research, there is no analogous database that can be used to advance the development of computational methods in each field. This is not because it wouldn't be useful, but rather because it is difficult to achieve for the reasons mentioned in section 1. We believe that these databases do not exist today because the software tools needed to generate these complicated databases and corresponding ETL pipelines does not exist. DBgen aims to fill that need.

5. Conclusions

In this work, we present DBgen, a python library that adds useful abstractions to the process of defining complex databases and ETL pipelines and thereby reduces the barrier to storing data in accordance with the FAIR principle. We also present a set of principles (SMART) that ETL pipelines should ideally abide by, analogous to the FAIR principles for data storage. We use materials science R&D data as an example of an inherently complex data source. We show that modeling the data without making assumptions demands a complicated database architecture, which would be difficult to create without DBgen. We show that modeling the data in this way adds transparency and flexibility to dataset curation. Altogether, we provide evidence that DBgen is a useful tool that greatly reduces the barrier to storing scientific data in accordance with the widely accepted FAIR principles.

6. Conflict of Interest

Dr. Brian Rohr and Dr. Michael Statt regularly use DBgen in their materials science data engineering services work at Modelyst, LLC.

References

[1] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne, et al., The fair guiding principles for scientific data management and stewardship, Scientific data 3 (1) (2016) 1–9.

[2] J. Wise, A. G. de Barron, A. Splendiani, B. Balali-Mood, D. Vasant, E. Little, G. Mellino, I. Harrow, I. Smith, J. Taubert, et al., Implementation and relevance of fair data principles in biopharmaceutical r&d, Drug discovery today 24 (4) (2019) 933–938.

[3] P. Deshpande, A. Rasin, J. Furst, D. Raicu, S. Antani, Diis: a biomedical data access framework for aiding data driven research supporting fair principles, Data 4 (2) (2019) 54.

[4] C. Vesteghem, R. F. Brøndum, M. Sønderkær, M. Sommer, A. Schmitz, J. S. Bødker, K. Dybkær, T. C. El-Galaly, M. Bøgsted, Implementing the fair data principles in precision oncology: review of supporting initiatives, Briefings in bioinformatics 21 (3) (2020) 936–945.

[5] T. Tanhua, S. Pouliquen, J. Hausman, K. O'brien, P. Bricher, T. de Bruin, J. J. Buck, E. F. Burger, T. Carval, K. S. Casey, et al., Ocean fair data services, Frontiers in Marine Science 6 (2019) 440.

[6] R. Devarakonda, G. Prakash, K. Guntupally, J. Kumar, Big federal data centers implementing fair data principles: Arm data center example, in: 2019 IEEE International Conference on Big Data (Big Data), IEEE, 2019, pp. 6033–6036.

[7] C. Gries, M. Servilla, M. O'brien, K. Vanderbilt, C. Smith, D. Costa, S. Grossman-Clarke, Achieving fair data principles at the environmental data initiative, the us-lter data repository, Biodiversity Information Science and Standards 3 (2019) e37047.

[8] L. Lannom, D. Koureas, A. R. Hardisty, Fair data and services in biodiversity science and geoscience, Data Intelligence 2 (1-2) (2020) 122–130.

[9] S. Calamai, F. Frontini, Fair data principles and their application to speech and oral archives, Journal of New Music Research 47 (4) (2018) 339–354.

[10] C. Pommier, C. Michotey, G. Cornut, P. Roumet, E. Duchêne, R. Flores, A. Lebreton, M. Alaux, S. Durand, E. Kimmel, et al., Applying

fair principles to plant phenotypic data management in gnpis, Plant Phenomics 2019 (2019).

[11] Y. Watanabe, K. F. Aoki-Kinoshita, Y. Ishihama, S. Okuda, Glycopost realizes fair principles for glycomics mass spectrometry data, Nucleic Acids Research 49 (D1) (2021) D1523–D1528.

[12] M. Boeckhout, G. A. Zielhuis, A. L. Bredenoord, The fair guiding principles for data stewardship: fair enough?, European journal of human genetics 26 (7) (2018) 931–936.

[13] C. Draxl, M. Scheffler, Big-data-driven materials science and its fair data infrastructure, arXiv preprint arXiv:1904.05859 (2019).

[14] R. David, L. Mabile, A. Specht, S. Stryeck, M. Thomsen, M. Yahia, C. Jonquet, L. Dollé, D. Jacob, D. Bailo, et al., Fairness literacy: the achilles' heel of applying fair principles, CODATA Data Science Journal 19 (32) (2020) 1–11.

[15] L. B. Da Silva Santos, M. D. Wilkinson, A. Kuzniar, R. Kaliyaperumal, M. Thompson, M. Dumontier, K. Burger, Fair data points supporting big data interoperability, Enterprise Interoperability in the Digitized and Networked Factory of the Future. ISTE, London (2016) 270–279.

[16] L. Garcia, J. Bolleman, S. Gehant, N. Redaschi, M. Martin, Fair adoption, assessment and challenges at uniprot, Scientific data 6 (1) (2019) 1–4.

[17] S. Stall, L. Yarmey, J. Cutcher-Gershenfeld, B. Hanson, K. Lehnert, B. Nosek, M. Parsons, E. Robinson, L. Wyborn, Make scientific data fair (2019).

[18] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, A. Lerer, Automatic differentiation in pytorch (2017).

[19] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker,

V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: Large-scale machine learning on heterogeneous systems, software available from tensorflow.org (2015).
URL `https://www.tensorflow.org/`

[20] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei, Imagenet: A large-scale hierarchical image database, in: 2009 IEEE conference on computer vision and pattern recognition, Ieee, 2009, pp. 248–255.

[21] A. Krizhevsky, I. Sutskever, G. E. Hinton, Imagenet classification with deep convolutional neural networks, Advances in neural information processing systems 25 (2012) 1097–1105.

[22] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.