LabbookDB: A Wet-Work-Tracking Database Application Framework

Horea-Ioan Ioanas[‡*], Bechara John Saab[§], Markus Rudin[‡]

https://youtu.be/yDBu_wSyw-g

Abstract—LabbookDB is a relational database application framework for life sciences—providing an extendable schema and functions to conveniently add and retrieve information, and generate summaries. The core concept of LabbookDB is that wet work metadata commonly tracked in lab books or spreadsheets is more efficiently and more reliably stored in a relational database, and more flexibly queried. We overcome the flexibility limitations of designed-for-analysis spreadsheets and databases by building our schema around atomized physical object interactions in the laboratory (and providing plotting- and/or analysis-ready dataframes as a compatibility layer). We keep our database schema more easily extendable and adaptable by using joined table inheritance to manage polymorphic objects and their relationships. LabbookDB thus provides a wet work metadata storage model excellently suited for explorative ex-post reporting and analysis, as well as a potential infrastructure for automated wet work tracking.

Index Terms—laboratory notebook, labbook, wet work, record keeping, internet of things, reports, life science, biology, neuroscience, behaviour, relational database, normalization, SQL

Introduction

The laboratory notebook (more commonly, lab book) is a long-standing multi-purpose record—serving as a primary data trace, as a calendar, diary, legal document, memory aid, organizer, timetable, and also proposed as a rapid science communication medium [Bra07]. It is of notable popularity in the natural sciences, especially in the life sciences—where research largely consists of "wet work" (i.e. real-world manipulation), which generally leaves no data trace unless explicitly recorded. With the advent of electronic data acquisition and storage, however, the lab book has increasingly lost significance as a repository for actual data, and has transformed into a metadata record. Notably, the modern lab book has become a general repository of information, for which simple array formats (e.g. tables, spreadsheets, or data matrices) do not provide an adequate input and/or storage format.

Some scientists and science service providers seek to emulate the seemingly convenient lab book format in the electronic medium—even providing support for sketching and doodling (e.g. eLabFTW [CNM12]). Storing information in free-text or pictorial form, however, exacerbates the incompatibility with electronic data analysis and reporting (which commonly requires consistent array formats). This approach, rather than merely retarding information flow by increasing the need for manual lookup and input, can also increase the incidence of biased evaluation—most easily as a consequence of notes being more often or more attentively consulted, and judged by varied but not explicitly documented standards, depending on the expectations of the researcher.

Conversely, researchers often force multidimensional and relationship-rich experimental metadata into the familiar and analysis-apt spreadsheet format. Under a list-like model, however, relationships become spread over many cell combinations while remaining untracked. This leads to information replication in multiple entries, which in turn renders e.g. the task of updating the correspondence between related cells non-trivial. These issues are known as information redundancy and update anomalies, respectively—and are prone to damage data integrity over time. The temptation also arises to truncate input to only what is considered essential at the time of the experiment. This runs the risk of omitting information which may have been easily recorded (even automatically) given a proper data structure, and which may become crucial for closer ex-post deliberation of results.

The crux of the issue, which neither of these approaches adequately addresses, is to store experimental metadata in a fashion which befits its relationship-rich nature, while providing array-formatted data output for analysis, and spreadsheet-formatted data for human inspection. Solutions which provide such functionality for a comprehensive experimental environment are few, and commonly proprietary and enterprise oriented (e.g. iRATS, REDCap [HTT+09]). One notable exception is MouseDB [Bri11], a database application framework built around mouse tracking. This package is considerably more mature than our present endeavour, yet more closely intended as a lab management tool rather than a general lab book replacement. It makes a number of differing structure choices, but given the permissive license (BSD [Ini99]) of both projects, it is conceivable for functionalities from one to be merged into another in the future.

The need for a wet work metadata system providing a better internal data model and consistently structured outputs, is compounded by the fact that such a system may also be better suited for (semi)automatic record keeping. Rudimentary semiautomatic tracking (via barcode-scanning) is already available for at least one commercial platform (iRATS), and the concept is likely to become of even more interest, as the internet of things reaches the laboratory. This makes a well-formed open source relational

* *Corresponding author: ioanas@biomed.ee.ethz.ch*
‡ *Institute for Biomedical Engineering, ETH and University of Zurich*
§ *Preclinical Laboratory for Translational Research into Affective Disorders, DPPP, Psychiatric Hospital, University of Zurich*

schema of object interactions accurately representing the physical world pivotal in the development of laboratory record keeping.

Methods

Database Management System

In order to cast complex laboratory metadata into tractable relationships with high enough entry numbers for statistical analysis, as well as in order to reduce data redundancy and the risk of introducing anomalies, we opt for a relational database management system, as interfaced with via SQLAlchemy. The scalability and input flexibility advantages of noSQL databases do not apply well to the content at hand, as experimental metadata is small, reliable, and slowly obtained enough to make scalability a secondary concern and schema quality and consistency a principal concern. Our robust but easily extendable schema design encapsulates contributors' wet work procedural knowledge, and is valuable in excess of creating an efficient storage model; as well-chosen predefined attributes facilitate reproducibility and encourage standardization in reporting and comparability across experiments.

Database Schema Design

The current database schema was generated from numerous bona fide spreadsheet formats used at the Psychiatry University Clinic, ETH, and University of Zurich. Iteratively, these spreadsheets are being normalized to first, second, third, and fourth normal forms (eliminating multivalued attributes, partial dependencies, transitive dependencies, and multivalued dependencies, respectively) [Cod74]. As the database schema of the current release (`0.0.1`) consists of over 40 tables, and is expected to expand as more facets of wet work are tracked, ensuring that relationships are well-formed will remain an ongoing process. The perpetually non-definitive nature of the database schema is also conditioned by the continuous emergence of new wet work methods.

Record Keeping and Structure Migration

We use version tracking via Git to provide both a verifiable primary input record, and the possibility to correct entries (e.g. typos) in order to facilitate later database usage in analysis. Version tracking of databases, however, is rendered difficult by their binary format. To mitigate this issue, as well as the aforementioned continuous structure update requirement, we track modular Python function calls which use the LabbookDB input application programming interface (API) to generate a database—instead of the database itself. We refer to this repository of Python function calls as the "source code" of the database.

Input Design

The LabbookDB input API consists of Python functions which interface with SQLAlchemy, and accept dictionary and string parameters for new entry specification and existing entry identification, respectively. These Python functions are wrapped for command line availability via `argh`—as sub-commands under the master command `LDB` in order to conserve executable namespace. Dictionaries are passed to the command line surrounded by simple quotes, and a LabbookDB-specific syntax was developed to make entry identification considerably shorter than standard SQL (though only arguably more readable).

Output Design

Outputs include simple human-readable command line reports and spreadsheets, `.pdf` protocols, introspective graphs, and dataframes. Dataframe output is designed to support both the Pandas `DataFrame` format and export as `.csv`. The dataframe conventions are kept simple and are perfectly understood by BehavioPy [Chr16], a collection of plotting functions originally developed as part of LabbookDB, but now branched off for more general usage. The formatting of command line reports is built by concatenating `__str__` methods of queryable objects and their immediate relationships, and is based on the most common use cases for rapid monitoring. Contingent on the availability of object-specific formatting guidelines, an interface is available for generating human-readable, itemized `.pdf` protocols.

Scope

To accommodate for a developing schema, reduce dependencies, and reduce usage difficulty, we opt to showcase LabbookDB as a personal database system, using SQLite as an engine. As such, the database is stored locally, managed without a client-server model, and accessed without the need for authentication. The scope thus extends to maximally a few users, which trust each other with full access. This is an appropriate scope for most research groups. Additionally, this design choice enables single researchers or clusters of researchers within a larger group to autonomously try out, test, contribute to, or adopt LabbookDB without significant overhead or the need for a larger institutional commitment.

Quality Control

LabbookDB provides an outline for unit testing which ships in the form of a submodule. Currently this is populated with a small number of simple example tests for low-level functionality, and is intended to grow as individual code units become more hardened. Additionally, we provide extensive integration testing which assures that the higher-level functionality of LabbookDB remains consistent, and that databases can be regenerated from updated source code as needed. The ever-increasing data required for extensive integration testing is distributed independently of LabbookDB and PIP, in a separate Git repository named Demolog [Chr17b]. Both unit and integration tests are currently run continuously with TravisCI.

Development Model

The database schema draws from ongoing input, testing, and the wet work experience of many researchers associated with the Institute of Biomedical Engineering and the Animal Imaging Center at the ETH and University of Zurich. The development team currently consists of one programmer (corresponding author), who will maintain and actively develop LabbookDB at least until 2019—independently of community involvement. Beyond that time point development may become contingent on the established impact of the project, including number of contributors, academic recognition of the metadata management system, adoption in the scientific Python or biomedical community, or the prospect of developing commercial tools to leverage the open source schema and API.

Documentation

Project documentation is published via Read the Docs, and contains a general project description, alongside installation instructions and a browsable listing of the API. The documentation model is based primarily on docstrings, but also contains example functions and example input stored in the corresponding submodule. A number of fully reproducible minimal input (working with the Demolog data only) versions of these functions are also presented in this paper.

Capabilities

The aforementioned integration testing data reposited as Demolog [Chr17b] demonstrates the capabilities of this first LabbookDB release in a concise fashion. Contingent on the presence of LabbookDB `0.0.1` [Chr17a] and its dependencies on the system, an example database can be built—and correspondingly described subsequent entries can be executed locally. To set up the example database, the following should be run from the terminal:

```
mkdir ~/src
cd ~/src
git clone https://bitbucket.org/TheChymera/demolog
cd demolog/from_python_code
./generate_db.py
mkdir ~/syncdata
cp meta.db ~/syncdata
```

Note that, for the examples to work, it is mandatory to create the `src` and `syncdata` directories under the user's home path.

Entry Insertion and Update

The Python API allows for clearly laid out entry insertion, via the `add_generic()` function:

```
add_generic(db_location, parameters={
    "CATEGORY":"Animal",
    "sex":"m",
    "ear_punches":"L",
    "license":"666/2013",
    "birth_date":"2016,7,21",
    "external_ids":[
        {"CATEGORY":"AnimalExternalIdentifier",
            "database":"ETH/AIC",
            "identifier":"5682",
            },
        {"CATEGORY":"AnimalExternalIdentifier",
            "database":"UZH/iRATS",
            "identifier":"M2889"
            },
        ],
    "genotypes":["Genotype:code.datg"],
    })
```

Technically, all entries could be created in such a fashion. However, in order to better organize logging (e.g. quarterly, as in the Demolog submodules), we provide an additional function for entry update. Instead of editing the original animal input file to set e.g. the death date, the animal entry can be updated via a separate function call:

```
append_parameter(db_location,
    entry_identification="Animal:external_ids."
        "AnimalExternalIdentifier:database."
        "ETH/AIC&#&identifier.5682",
    parameters={
        "death_date":"2017,5,13,17,25",
        "death_reason":"end of experiment",
        }
    )
```

In this example an existing entry is selected in a compact fashion using custom LabbookDB syntax.

Compact Syntax for Entry Selection

In order to compactly identify related for data input, we have developed a custom LabbookDB syntax. This syntax is automatically parsed by the `labbookdb.db.add.get_related_ids()` function, which is called internally by input functions. Notably, understanding of this syntax is not required in order to use reporting functions, and plenty of examples of its usage for input can be seen in Demolog.

Custom LabbookDB syntax is not written as a wrapper for SQL, but rather specifically designed to satisfy LabbookDB entry selection use cases in a minimum number of characters. This is primarily provided to facilitate database manipulation from the command line, though it also aids in making database source code more clearly laid out

Consider the string used to identify the entry to be updated in the previous code snippet (split to fit document formatting):

```
"Animal:external_ids.AnimalExternalIdentifier:datab"
"ase.ETH/AIC&#&identifier.5682"
```

Under the custom LabbookDB syntax, the selection string always starts with the entry's object name (in the string at hand, `Animal`). The object name is separated from the name of the attribute to be matched by a colon, and the attribute name is separated from the value identifying the existing entry by a period. The value can be either a string, or—if the string contains a colon—it is presumed to be another object (which is then selected using the same syntax). Multiple matching constraints can be specified, by separating them via double ampersands. Inserting one or multiple hashtags in between the ampersands indicates at what level the additional constraint is to be applied. In the current example, two ampersands separated by one hashtag mean that an `AnimalExternalIdentifier` object is matched contingent on a `database` attribute value of `"ETH/AIC"` and an `identifier` attribute value of `"5682"`. Had the ampersands not been separated by a hashtag, the expression would have prompted LabbookDB to apply the additional `identifier` attribute constraint not to the `AnimalExternalIdentifier` object, but one level higher, to the `Animal` object.

Command Line Reporting

Quick reports can be generated directly via the command line, e.g. in order to get the most relevant aspects of an animal at a glance. The following code should be executable locally in the terminal, contingent on LabbookDB example database availability:

```
LDB animal-info -p ~/syncdata/meta.db 5682 ETH/AIC
```

The code should return an overview similar to the following, directly in the terminal:

```
Animal(id: 15, sex: m, ear_punches: L):
   license: 666/2013
   birth:   2016-07-21
   death:   2017-05-13 (end of experiment)
   external_ids:    5682(ETH/AIC), M2889(UZH/iRATS)
   genotypes:       DAT-cre(tg)
   cage_stays:
      cage 31, starting 2016-12-06
      cage 37, starting 2017-01-10
   operations:
      Operation(2017-03-04 10:30:00: virus_injection)
      Operation(2017-03-20 13:00:00: optic_implant)
   treatments:
   measurements:
      Weight(2016-12-22 13:35:00, weight: 29.6g)
      Weight(2017-03-30 11:48:00, weight: 30.2g)
      fMRI(2016-12-22 13:35:49, temp: 35.0)
```

```
fMRI(2017-03-30 11:48:52, temp: 35.7)
Weight(2017-04-11 12:33:00, weight: 29.2g)
fMRI(2017-04-11 12:03:58, temp: 34.8)
Weight(2017-05-13 16:53:00, weight: 29.2g)
```

Human Readable Spreadsheets

LabbookDB can join tables from the database in order to construct comprehensive human-readable spreadsheet overviews. Storing information in a well-formed relational structure allows for versatile and complex reporting formats. In the following model, for instance, the "responsive functional measurements" column is computed automatically from the number of fMRI measurements and the number of occurrences of the `"ICA failed to indicate response to stimulus"` irregularity on these measurements.

Contingent on the presence of LabbookDB and the example database, the following lines of code should generate a dataframe formatted in the same fashion as Table 1, and return it directly in the terminal, or save it in `.html` format, respectively:

```
LDB animals-info ~/syncdata/meta.db
LDB animals-info ~/syncdata/meta.db -s overview
```

An example of the `.html` output can be seen in the Demolog repository under the `outputs` directory.

Printable Protocol Output

LabbookDB can create `.pdf` outputs to serve as portable step-by-step instructions suitable for computer-independent usage. This capability, paired with the database storage of e.g. protocol parameters means that one can store and assign very many protocol variants internally (with a minuscule storage footprint), and conveniently print out a preferred protocol for collaborators, technicians, or students, without encumbering their workflow with any unneeded complexity. The feature can be accessed from the `labbookdb.report.examples` module. The following code should be executable locally, contingent on LabbookDB and example database availability:

```
from labbookdb.report.examples import protocol

class_name = "DNAExtractionProtocol"
code = "EPDqEP"
protocol("~/syncdata/meta.db", class_name, code)
```

This should create a `DNAExtractionProtocol_EPDqEP.pdf` file identical to the one tracked in Demolog.

Introspection

LabbookDB ships with a module which generates graphical representations of the complex relational structures implemented in the package. The feature is provided by the `labbookdb.introspection.schema` module. The following code should be executable locally, contingent on LabbookDB availability:

```
from labbookdb.introspection.schema import generate

extent=[
    "Animal",
    "FMRIMeasurement",
    "OpenFieldTestMeasurement",
    "WeightMeasurement",
    ]
save_plot = "~/measurements_schema.pdf"
generate(extent, save_plot=save_plot)
```

This example should generate Figure 1 in `.pdf` format (though `.png` is also supported).

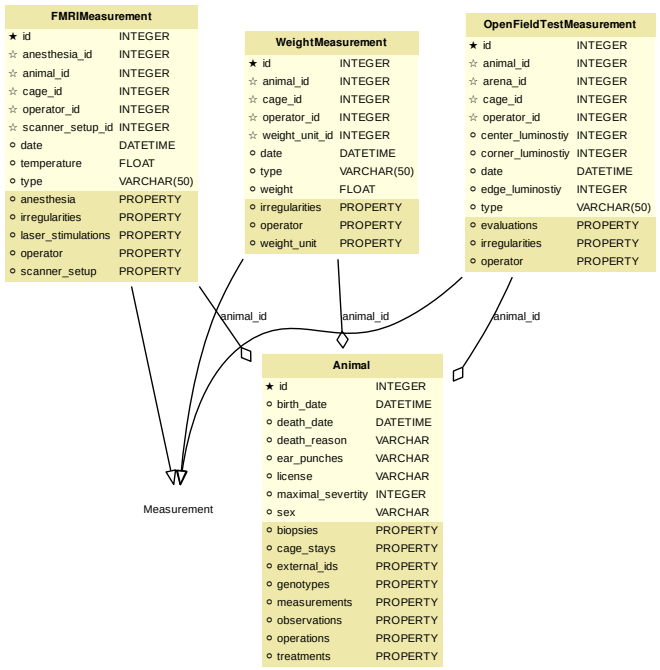

Fig. 1: *LabbookDB schema section, illustrating the polymorphic relationship between Animal objects and different Measurement variants.*

Polymorphic Mapping and Schema Extension

In current research, it is common to subject animals to experimental procedures which are similar in kind, but which can be split into categories with vastly different attributes. Prime examples of such procedures are `Measurements` and `Operations`. In Figure 1 we present how LabbookDB uses SQLAlchemy's joined table inheritance to link different measurement types to the `measurements` attribute of the `Animal` class. Attributes common to all measurement types are stored on the `measurements` table, as are relationships common to multiple measurements (e.g. the relationship to the `Animal` class, instantiated in the `animal_id` attribute).

One of the foremost requirements for a relational database application to become a general purpose lab book replacement is an easily extendable schema. The `Measurement` and `Operation` base classes demonstrate how inheritance and polymorphic mapping can help extend the schema to cover new types of work without changing existing classes. Polymorphism can be extended to more classes, to further propagate this feature. For instance, all measurement subjects in LabbookDB databases are currently recorded as `Animal` objects. This is adequate for most rodents, however it remains inadequate for e.g. human subjects. The issue would best be resolved by creating a `Subject` class, with attributes (including relationships) common to multiple types of subjects, and then creating derived classes, such as `HumanSubject` or `MouseSubject` to track more specific attributes. `Measurement` and `Operation` assignments would be seamlessly transferable, as relationships between objects derived from the `Subject` base class and e.g. the `Operation` base class would be polymorphic.

Animal_id	ETH/AIC	UZH/iRATS	Genotype_code	Animal_death_date	responsive functional measurements
45	6258	M5458	datg	2017-04-20 18:30:00	0/0
44	6262	M4836	eptg	None	2/2
43	6261	M4835	eptg	2017-04-09 18:35:00	0/0
42	6256	M4729	epwt	None	0/0
41	6255	M4728	eptg	None	2/2

TABLE 1: *Example of a human-readable overview spreadsheet generated via the LabbookDB command line functionality.*

Atomized Relationships

We use the expression "atomized relationships" to refer to the finest grained representation of a relationship which can feasibly be observed in the real world. In more common relational model terms, higher atomization would correspond to higher normal forms—though we prefer this separate nomenclature to emphasize the preferential consideration of physical interactions, with an outlook towards more easily automated wet work tracking. Similarly to higher normal forms, increasingly atomized relationships give rise to an increasingly complex relational structure of objects with decreasing numbers of attributes. LabbookDB embraces the complexity thus generated and the flexibility and exploratory power it facilitates. Database interaction in LabbookDB is by design programmatic, an thus ease of human readability of the raw relational structure is only of subordinate concern to reporting flexibility.

An example of relationship atomization is showcased in Figure 2. Here the commonplace one-to-many association between `Cage` and `Animal` objects is replaced by a `CageStay` junction table highlighting the fact that the relationship between `Cage` and `Animal` is bounded by time, and that while it is many-to-one at any one time point, in the overarching record it is, in fact, many-to-many. This structure allows animals to share a cage for a given time frame, and to be moved across cages independently—reflecting the physical reality in animal housing facilities. This complexity is seamlessly handled by LabbookDB reporting functions, as seen e.g. in the command line reporting example previously presented.

Conversely, atomization can result in a somewhat simpler schema, as higher level phenomena may turn out to be special cases of atomized interactions. By design (and in contrast to the MouseDB implementation), we would not track breeding cages as a separate entity, as the housing relationships are not distinct from those tracked by the `CageStay` object. A separate object may rather be introduced for breeding events—which need not overlap perfectly with breeding cages.

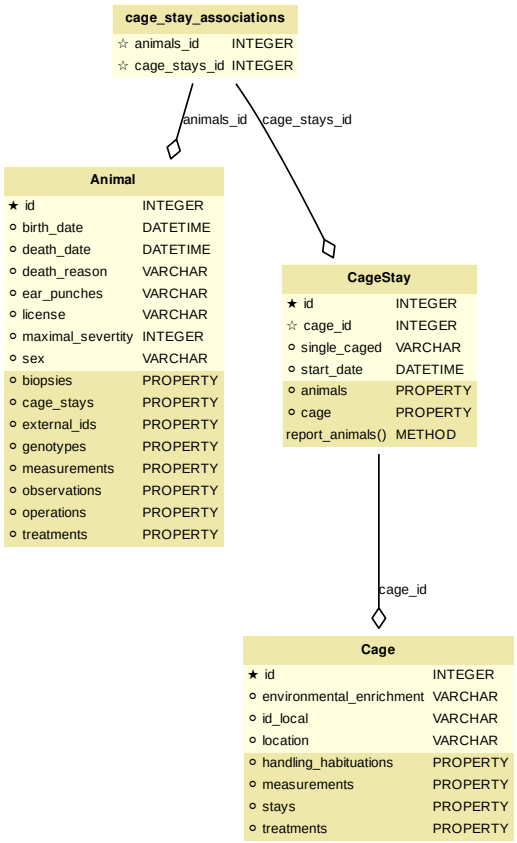

Fig. 2: *LabbookDB schema section, illustrating a more complex and accurate representation of the relational structure linking animals and cages in the housing facility.*

Irregularity and Free Text Management

The atomized schema seeks to introduce structure wherever possible, but also provides a bare minimum set of free-text fields, to record uncategorizable occurrences. Irregular events associated with e.g. `Measurement` or `Operation` instances are stored in the `irregularities` table, and linked by a many-to-many relationship to the respective objects. This not only promotes irregularity re-use, but also facilitates rudimentary manual pattern discovery, and the organic design of new objects within the schema.

Irregular events can also be recorded outside of predetermined interventions, via `Observation` objects. These objects have their own date attribute, alongside free-text attributes, and a `value` attribute, to more appropriately record a quantifiable trait in the observation.

Plotting via BehavioPy

LabbookDB provides a number of powerful data selection and processing functions, which produce consistently structured dataframes that seamlessly integrate with the BehavioPy [Chr16] plotting API. The forced swim test, for instance, is a preclinically highly relevant behavioural assay [PDCB05], which LabbookDB can document and evaluate. The following example code should be executable locally, contingent on LabbookDB, example database, and example data (included in Demolog) availability:

ID	Immobility Ratio	Interval [1 min]	Treatment
28	0.2635	3	Control
28	0.1440	2	Control
30	0.6813	3	Control
1	0.6251	6	Fluoxetine
32	0.6695	5	Fluoxetine
2	0.6498	6	Fluoxetine

TABLE 2: *Example of LabbookDB processed data output for the forced swim test. The format precisely matches the requirements of BehavioPy plotting functions.*

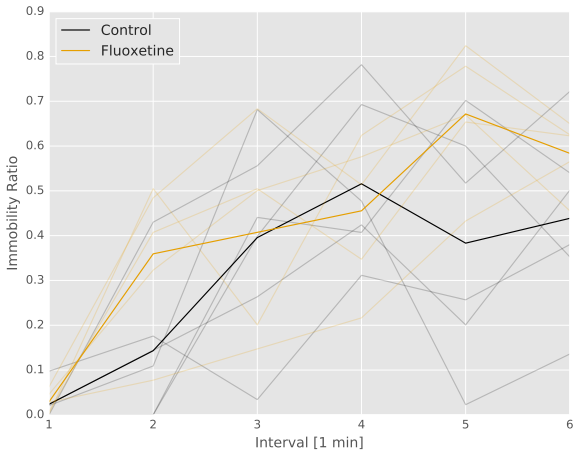

Fig. 3: *Timecourse plot of the forced swim test performed on mice in different treatment groups—automatically generated by LabbookDB, using plotting bindings from BehavioPy.*

```
import matplotlib.pyplot as plt
from labbookdb.report.behaviour import forced_swim

start_dates = ["2017,1,31,22,0","2016,11,24,21,30"]
forced_swim("~/syncdata/meta.db", "tsplot",
    treatment_start_dates=start_dates
    save_df="~/fst_df.csv")
plt.show()
```

The above code prompts LabbookDB to traverse the complex relational structure depicted in Figure 4, in order to join the values relevant to evaluation of the forced swim test. `Animal` objects are joined to `Treatment.code` values via their relationships to `Cage` and `CageStay` objects. This relational structure is determined by the administration of drinking water treatments at the cage level, and thus their contingence on the presence of animals in cages at the time of the treatment. Further, `Evaluation.path` values are joined to `Animal` objects (via their respective relationships to `Measurement` objects) in order to determine where the forced swim test evaluation data is stored for every animal. Subsequently, the annotated event tracking data is processed into desired length time bins (here, 1 minute), and immobility ratios are calculated per bin. Finally, the data is cast into a consistent and easily readable dataframe (formatted in the same fashion as Table 2) which can be both saved to disk, or passed to the appropriate BehavioPy plotting function, to produce Figure 3.

Discussion and Outlook

Record Keeping

Version tracking of database generation source code adequately addresses the main record keeping challenges at this stage of the project. Additionally, it has a number of secondary benefits, such as providing comprehensive and up-to-date usage examples. Not least of all, this method provides a very robust backup—as the database can always be rebuilt from scratch. A very significant drawback of this approach, however, is poor scalability.

As the amount of metadata reposited in a LabbookDB database increases, the time needed for database re-generation may reach unacceptable levels. Disk space usage, while of secondary concern, may also become an issue. Going forward, better solutions for record keeping should be implemented.

Of available options we would preferentially consider input code tracking (if possible in a form which is compatible with incremental execution) rather than output code tracking (e.g. in the form of data dumps). This is chiefly because output code tracking would be dependent not only of the data being tracked, but also of the version of LabbookDB used for database creation—ideally these versioning schemes would not have to become convoluted.

Structure Migration

The long-term unsustainability of database source code tracking also means that a more automated means of structure migration should be developed, so that LabbookDB databases can be recast from older relational structures into improved and extended newer structures—instead of relying on source code editing and regeneration from scratch. Possibly, this could be handled by shipping an update script with every release—though it would be preferable if this could be done in a more dynamic, rolling release fashion.

Data Input

Data input via sequential Python function calls requires a significant amount of boilerplate code, and appears very intransparent for users unaccustomed to the Python syntax. It also requires interfacing with an editor, minding syntax and formatting conventions, and browsing directory trees for the appropriate file in which to reposit the function calls.

While LabbookDB provides a command line interface to input the exact same data with the exact same dictionary and string conventions with arguably less boilerplate code, this input format has not been implemented for the full database generation source code. The main concern precluding this implementation is that the syntax, though simplified form standard SQL, is not nearly simple enough to be relied on for the robustness of thousands of manual input statements generated on-site.

A better approach may be to design automated recording workflows, which prompt the researcher for values only, while applying structure internally, based on a number of templates. Another possibility would be to write a parser for spreadsheets, which applies known LabbookDB input structures, and translates them into the internal relational representation. This second approach would also benefit from the fact that spreadsheets are already a very popular way in which researchers record their metadata—and could give LabbookDB the capability to import large numbers of old records, with comparatively little manual intervention.

Not least of all, the ideal outlook for LabbookDB is to automatically handle as much of the data input process as possible,

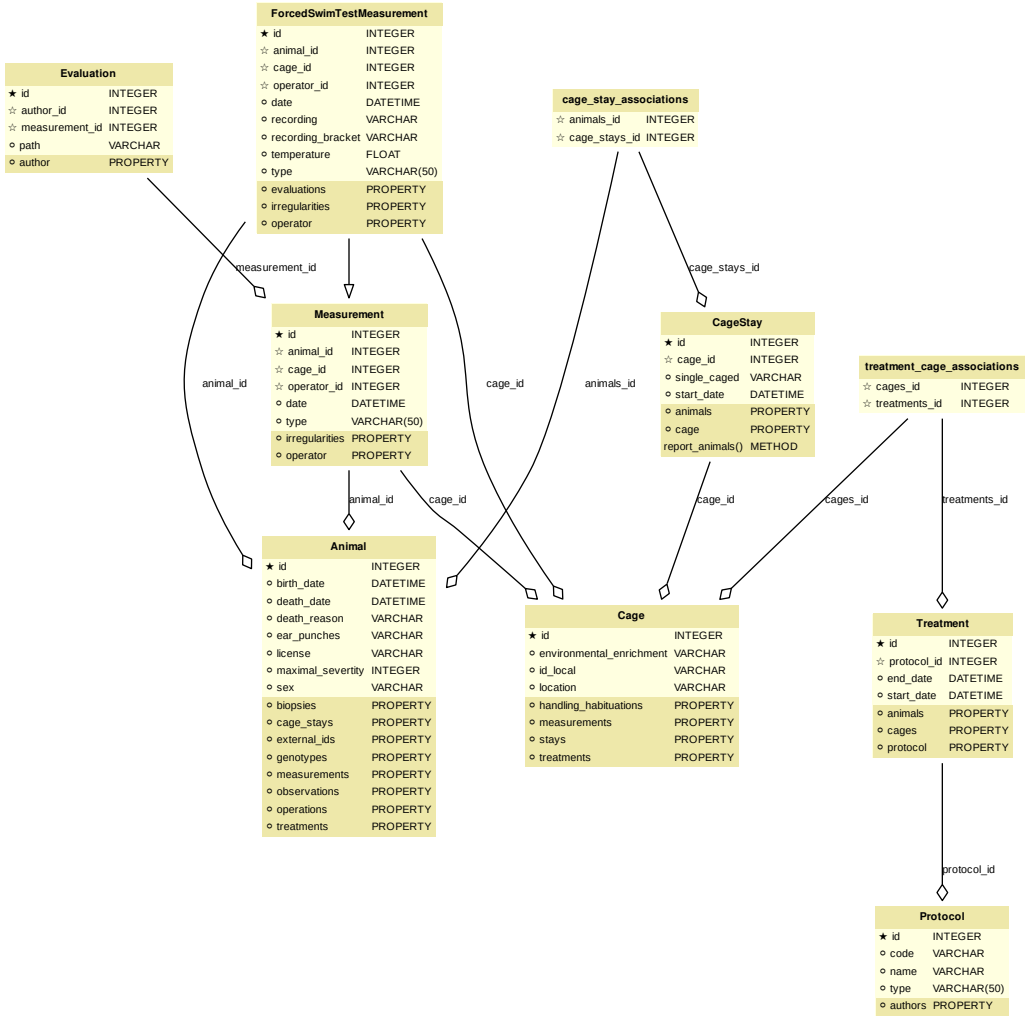

Fig. 4: *LabbookDB schema section relevant for constructing a plottable forced swim test dataframe.*

e.g. via specialized sensors, via semantic image [YJW+16] or video evaluation, or via an entity-barcode-scanner (as currently used by the iRATS system) . This poses nontrivial engineering challenges in excess of relation modelling, and requires distinctly more manpower than currently available. However, LabbookDB is from the licensing point of view suitable for use in commercial products, and additional manpower may be provided by science service providers interested in offering powerful, transparent, and extendable metadata tracking to their discerning customers.

Graphical User Interface

A notable special case of data input is the graphical user interface (GUI). While we acknowledge the potential of a GUI to attract scientists who are not confident users of the command line, we both believe that such an outreach effort is incompatible with the immediate goals of the project and that it is not typically an attractive long-term outlook for scientific Python applications.

Particularly at this stage in development, manpower is limited, and contributions are performed on a per-need basis (little code was written which was not relevant to addressing an actual data management issue). Presently our foremost outreach target are researchers who possess the technical affinity needed to test our schema at its fringes and contribute to—or comment on—our code and schema. A GUI would serve to add further layers of abstraction and make it more difficult for users to provide helpful feedback in our technology development efforts.

In the long run, we would rather look towards developing more automatic or implicit tracking of wet work, rather than simply writing a GUI. Our outlook towards automation also means that a GUI is likely to remain uninteresting for the use cases of the developers themselves, which would make the creation of such an interface more compatible with a commercial service model than with the classical Free and Open Source user-developer model.

REFERENCES

[Bra07] Jean-Claude Bradley. Open notebook science using blogs and wikis. 2007.

[Bri11] Dave Bridges. Mousedb. GitHub, 2011. URL: https://github.com/davebridges/mousedb.

[Chr16] Horea Christian. Behaviopy - behavioural data analysis and plotting in python. GitHub, 2016. URL: https://github.com/TheChymera/behaviopy, doi:10.5281/zenodo.188169.

[Chr17a] Horea Christian. Labbookdb - lab book database schema with information addition, retrieval, and reporting functions. GitHub, 2017. URL: https://github.com/TheChymera/LabbookDB, doi:10.5281/zenodo.823366.

[Chr17b] Horea Christian. Logging examples for labbookdb for scipy2017 proceedings, 05 2017. URL: https://bitbucket.org/TheChymera/demolog/src/9ce8ca3b808259a1cfe74169d7a91fb40e4cfd90?at=master.

[CNM12] Nicolas Carpi, Pascal Noirci, and Alexander Minges. elabftw. online, 2012. URL: https://www.elabftw.net/.

[Cod74] Edgar F. Codd. Recent investigations into relational data base systems. Technical Report RJ1385, IBM, 4 1974.

[HTT+09] Paul A Harris, Robert Taylor, Robert Thielke, Jonathon Payne, Nathaniel Gonzalez, and Jose G Conde. Research electronic data capture (redcap)—a metadata-driven methodology and workflow process for providing translational research informatics support. *Journal of biomedical informatics*, 42(2):377–381, 2009.

[Ini99] Open Source Initiative. The 3-clause bsd license. online, 07 1999. URL: https://opensource.org/licenses/BSD-3-Clause.

[PDCB05] Benoit Petit-Demouliere, Franck Chenu, and Michel Bourin. Forced swimming test in mice: a review of antidepressant activity. *Psychopharmacology*, 177(3):245–255, 2005.

[YJW+16] Quanzeng You, Hailin Jin, Zhaowen Wang, Chen Fang, and Jiebo Luo. Image captioning with semantic attention. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.