DataJoint User Manual

Release python-v0.12

DataJoint contributors

Feb 28, 2021

TABLE OF CONTENTS

This is a detailed manual for active users of DataJoint in Python.

This documentation can be read sequentially from start to end or used as reference for specific topics.

For a guided introduction to DataJoint, please explore our tutorials at http://tutorials.datajoint.io

CHAPTER

ONE

INTRODUCTION

1.1 Data Pipelines

1.1.1 What is a data pipeline?

A scientific **data pipeline** is a collection of processes and systems for organizing the data, computations, and workflows used by a research group as they jointly perform complex sequences of data acquisition, processing, and analysis.

A variety of tools can be used for supporting shared data pipelines:

Data repositories Research teams set up a shared **data repository**. This minimal data management tool allows depositing and retrieving data and managing user access. For example, this may include a collection of files with standard naming conventions organized into folders and sub-folders. Or a data repository might reside on the cloud, for example in a collection of S3 buckets. This image of data management – where files are warehoused and retrieved from a hierarchically-organized system of folders – is an approach that is likely familiar to most scientists.

Database systems **Databases** are a form of data repository providing additional capabilities:

1) Defining, communicating, and enforcing structure in the stored data.
2) Maintaining data integrity: correct identification of data and consistent cross-references, dependencies, and groupings among the data.
3) Supporting queries that retrieve various cross-sections and transformation of the deposited data.

Most scientists have some familiarity with these concepts, for example the notion of maintaining consistency between data and the metadata that describes it, or applying a filter to an Excel spreadsheet to retrieve specific subsets of information. However, usually the more advanced concepts involved in building and using relational databases fall under the specific expertise of data scientists.

Data pipelines **Data pipeline** frameworks may include all the features of a database system along with additional functionality:

1) Integrating computations to perform analyses and manage intermediate results in a principled way.
2) Supporting distributed computations without conflict.
3) Defining, communicating, and enforcing **workflow**, making clear the sequence of steps that must be performed for data entry, acquisition, and processing.

Again, the informal notion of an analysis "workflow" will be familiar to most scientists, along with the logistical difficulties associated with managing a workflow that is shared by multiple scientists within or across labs.

Therefore, a full-featured data pipeline framework may also be described as a scientific workflow system.

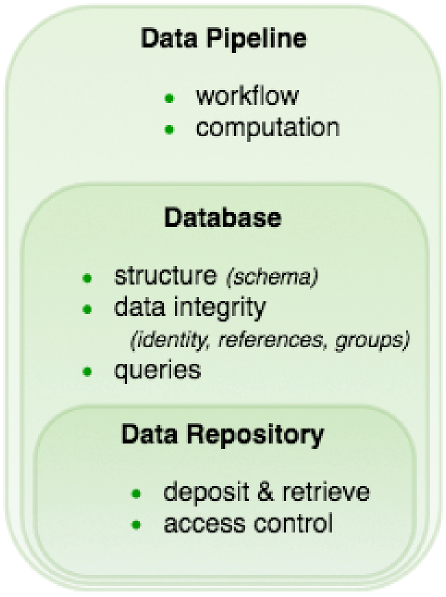


Fig. 1: Major features of data management frameworks: data repositories, databases, and data pipelines.

1.1.2 What is DataJoint?

DataJoint is a free open-source framework for creating scientific data pipelines directly from MATLAB or Python (or any mixture of the two). The data are stored in a language-independent way that allows interoperability between MATLAB and Python, with additional languages in the works. DataJoint pipelines become the central tool in the operations of data-intensive labs or consortia as they organize participants with different roles and skills around a common framework.

In DataJoint, a data pipeline is a sequence of steps (more generally, a directed acyclic graph) with integrated data storage at each step. The pipeline may have some nodes requiring manual data entry or import from external sources, some that read from raw data files, and some that perform computations on data stored in other database nodes. In a typical scenario, experimenters and acquisition instruments feed data into nodes at the head of the pipeline, while downstream nodes perform automated computations for data processing and analysis.

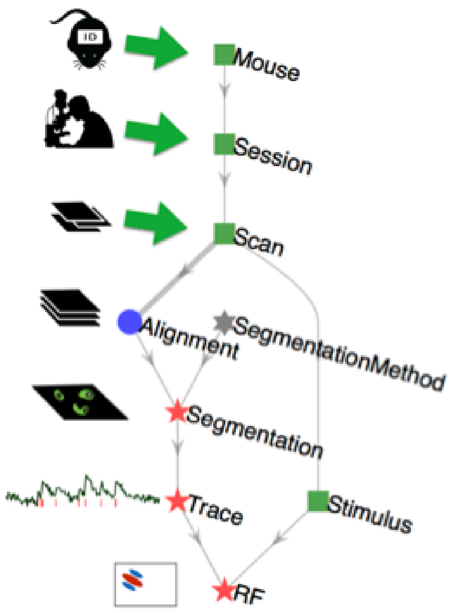


Fig. 2: For example, this is the pipeline for a simple mouse experiment involving calcium imaging in mice.

In this example, the experimenter first enters information about a mouse, then enters information about each imaging session in that mouse, and then each scan performed in each imaging session. Next the automated portion of the pipeline takes over to import the raw imaging data, perform image alignment to compensate for motion, image segmentation to identify cells in the images, and extraction of calcium traces. Finally, the receptive field (RF) computation is performed by relating the calcium signals to the visual stimulus information.

1.1.3 How DataJoint works

DataJoint enables data scientists to build and operate scientific data pipelines.

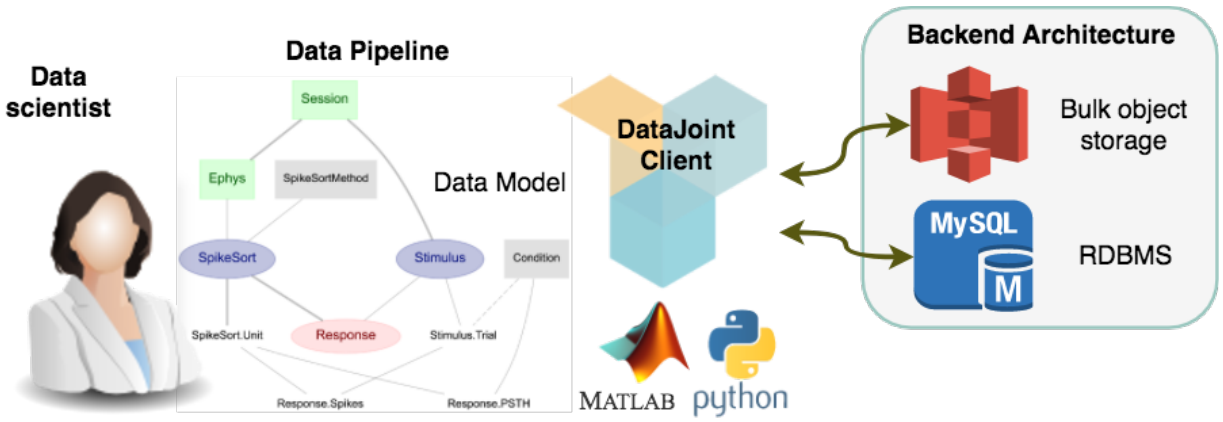


Fig. 3: Conceptual overview of DataJoint operation.

DataJoint provides a simple and powerful data model, which is detailed more formally in Yatsenko D, Walker EY, Tolias AS (2018). DataJoint: A Simpler Relational Data Model.. Put most generally, a "data model" defines how to think about data and the operations that can be performed on them. DataJoint's model is a refinement of the relational data model: all nodes in the pipeline are simple tables storing data, tables are related by their shared attributes, and query operations can combine the contents of multiple tables. DataJoint enforces specific constraints on the relationships between tables that help maintain data integrity and enable flexible access. DataJoint uses a succinct data definition language, a powerful data query language, and expressive visualizations of the pipeline. A well-defined and principled approach to data organization and computation enables teams of scientists to work together efficiently. The data become immediately available to all participants with appropriate access privileges. Some of the "participants" may be computational agents that perform processing and analysis, and so DataJoint features a built-in distributed job management process to allow distributing analysis between any number of computers.

From a practical point of view, the back-end data architecture may vary depending on project requirements. Typically, the data architecture includes a relational database server (e.g. MySQL) and a bulk data storage system (e.g. AWS S3 or a filesystem). However, users need not interact with the database directly, but via MATLAB or Python objects that are each associated with an individual table in the database. One of the main advantages of this approach is that DataJoint clearly separates the data model facing the user from the data architecture implementing data management and computing. DataJoint works well in combination with good code sharing (e.g. with git) and environment sharing (e.g. with Docker)

DataJoint is designed for quick prototyping and continuous exploration as experimental designs change or evolve. New analysis methods can be added or removed at any time, and the structure of the workflow itself can change over time, for example as new data acquisition methods are developed.

With DataJoint, data sharing and publishing is no longer a separate step at the end of the project. Instead data sharing is an inherent feature of the process: to share data with other collaborators or to publish the data to the world, one only needs to set the access privileges.

1.1.4 Real-life example

The Mesoscale Activity Project (MAP) is a collaborative project between four neuroscience labs. MAP uses DataJoint for data acquisition, processing, analysis, interfaces, and external sharing.

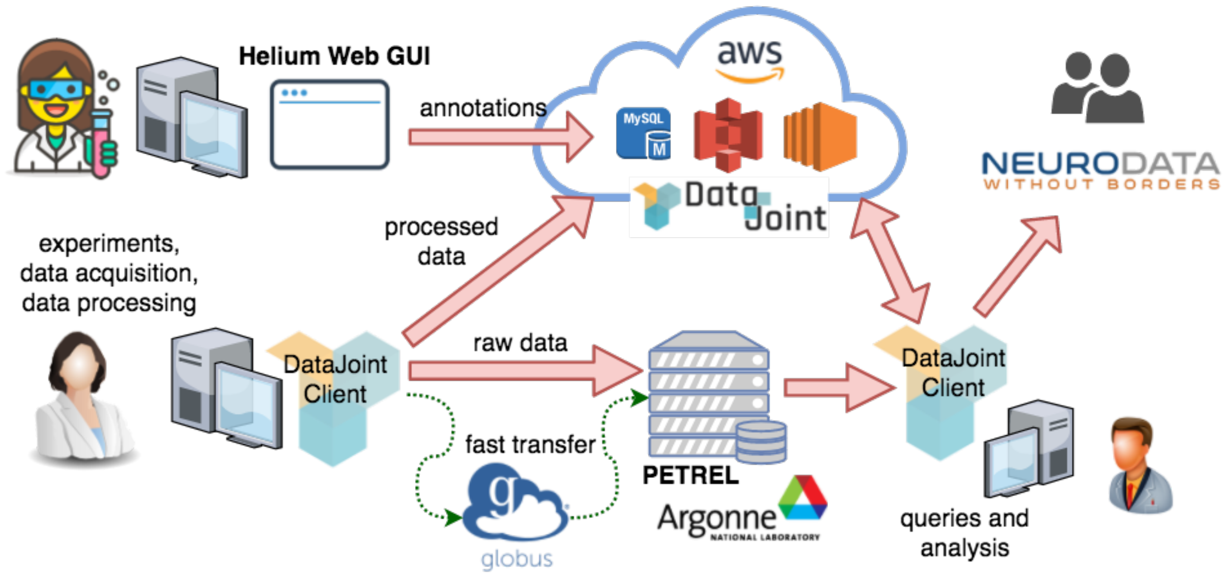


Fig. 4: The DataJoint pipeline for the MAP project.

The pipeline is hosted in the cloud through Amazon Web Services (AWS). MAP data scientists at the Janelia Research Campus and Baylor College of Medicine defined the data pipeline. Experimental scientists enter manual data directly into the pipeline using the Helium web interface. The raw data are preprocessed using the DataJoint client libraries in MATLAB and Python; the preprocessed data are ingested into the pipeline while the bulky and raw data are shared using Globus transfer through the PETREL storage servers provided by the Argonne National Lab. Data are made immediately available for exploration and analysis to collaborating labs, and the analysis results are also immediately shared. Analysis data may be visualized through web interfaces. Intermediate results may be exported into the NWB format for sharing with external groups.

1.1.5 Summary of DataJoint features

1. A free, open-source framework for scientific data pipelines and workflow management
2. Data hosting in cloud or in-house
3. MySQL, filesystems, S3, and Globus for data management
4. Define, visualize, and query data pipelines from MATLAB or Python
5. Enter and view data through GUIs
6. Concurrent access by multiple users and computational agents
7. Data integrity: identification, dependencies, groupings
8. Automated distributed computation

1.2 Teamwork

1.2.1 Data management in a science project

Science labs organize their projects as a sequence of activities of experiment design, data acquisition, and processing and analysis.

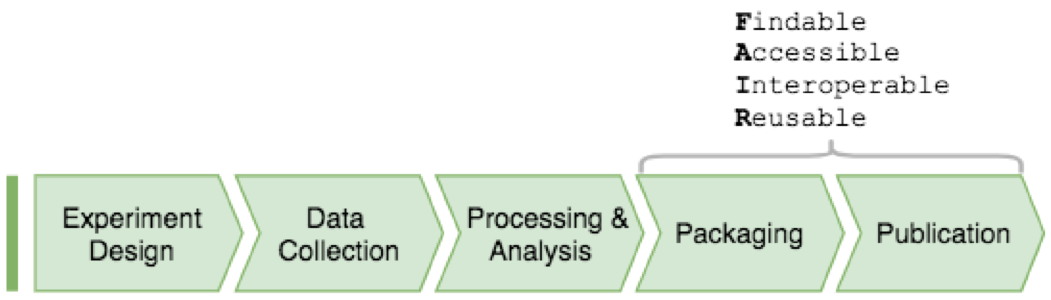


Fig. 5: Workflow and dataflow in a common findings-centered approach to data science in a science lab.

Many labs lack a uniform data management strategy that would span longitudinally across the entire project lifecycle as well as laterally across different projects.

Prior to publishing their findings, the research team may need to publish the data to support their findings. Without a data management system, this requires custom repackaging of the data to conform to the FAIR principles for scientific data management.

1.2.2 Data-centric project organization

DataJoint is designed to support a data-centric approach to large science projects in which data are viewed as a principal output of the research project and are managed systematically throughout in a single framework through the entire process.

This approach requires formulating a general data science plan and upfront investment for setting up resources and processes and training the teams. The team uses DataJoint to build data pipelines to support multiple projects.

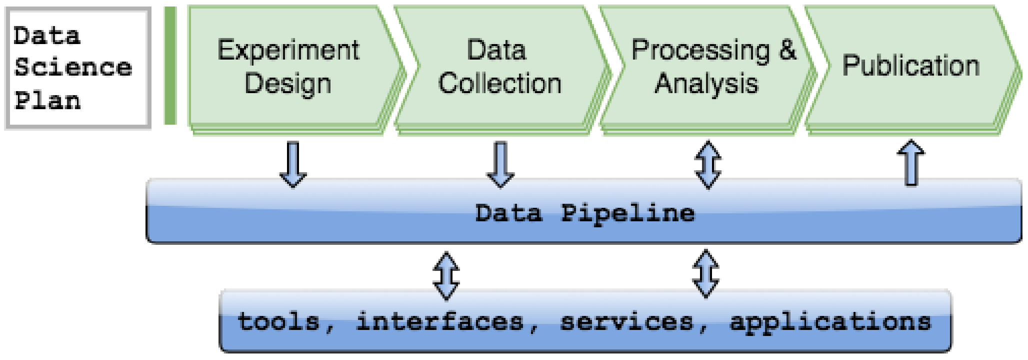


Fig. 6: Workflow and dataflow in a data pipeline-centered approach.

Data pipelines support project data across their entire lifecycle, including the following functions

- experiment design
- animal colony management
- electronic lab book: manual data entry during experiments through graphical user interfaces.
- acquisition from instrumentation in the course of experiments
- ingest from raw acquired data
- computations for data analysis
- visualization of analysis results
- export for sharing and publishing

Through all these activities, all these data are made accessible to all authorized participants and distributed computations can be done in parallel without compromising data integrity.

1.2.3 Team roles

The adoption of a uniform data management framework allows separation of roles and division of labor among team members, leading to greater efficiency and better scaling.

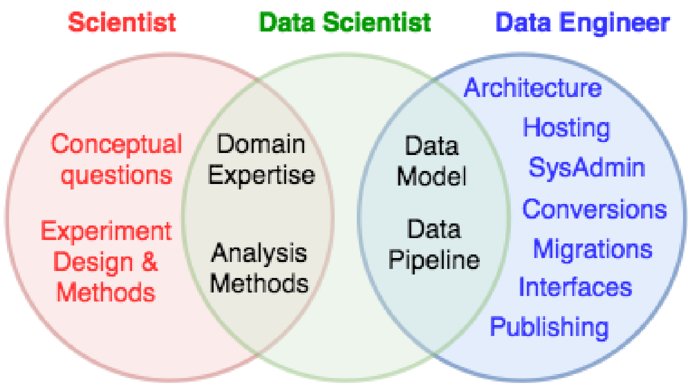


Fig. 7: Distinct responsibilities of data science and data engineering.

Scientists

design and conduct experiments, collecting data. They interact with the data pipeline through graphical user interfaces designed by others. They understand what analysis is used to test their hypotheses.

Data scientists

have the domain expertise and select and implement the processing and analysis methods for experimental data. Data scientists are in charge of defining and managing the data pipeline using DataJoint's data model, but they may not know the details of the underlying architecture. They interact with the pipeline using client programming interfaces directly from languages such as MATLAB and Python.

The bulk of this manual is written for working data scientists, except for System Administration.

Data engineers

work with the data scientists to support the data pipeline. They rely on their understanding of the DataJoint data model to configure and administer the required IT resources such as database servers, data storage servers, networks, cloud instances, Globus endpoints, etc. Data engineers can provide general solutions such as web hosting, data publishing, interfaces, exports and imports.

The System Administration section of this tutorial contains materials helpful in accomplishing these tasks.

DataJoint is designed to delineate a clean boundary between **data science** and **data engineering**. This allows data scientists to use the same uniform data model for data pipelines backed by a variety of information technologies. This delineation also enables economies of scale as a single data engineering team can support a wide spectrum of science projects.

1.3 Input and Output

1.3.1 Where are my data?

New users often ask this question thinking of passive **data repositories** – collections of files and folders and a separate collection of metadata – information about how the files were collected and what they contain. Let's address metadata first, since the answer there is easy: Everything goes in the database! Any information about the experiment that would normally be stored in a lab notebook, in an Excel spreadsheet, or in a Word document is entered into tables in the database. These tables can accommodate numbers, strings, dates, or numerical arrays. The entry of metadata can be manual, or it can be an automated part of data acquisition (in this case the acquisition software itself is modified to enter information directly into the database).

Depending on their size and contents, raw data files can be stored in a number of ways. In the simplest and most common scenario, raw data continue to be stored in either a local filesystem or in the cloud as collections of files and folders. The paths to these files are entered in the database (again, either manually or by automated processes). This is the point at which the notion of a **data pipeline** begins. Below these "manual tables" that contain metadata and file paths are a series of tables that load raw data from these files, process it in some way, and insert derived or summarized data directly into the database. For example, in an imaging application, the very large raw .TIFF stacks would reside on the filesystem, but the extracted fluorescent trace timeseries for each cell in the image would be stored as a numerical array directly in the database. Or the raw video used for animal tracking might be stored in a standard video format on the filesystem, but the computed X/Y positions of the animal would be stored in the database. Storing these intermediate computations in the database makes them easily available for downstream analyses and queries.

1.3.2 Do I have to manually enter all my data into the database?

No! While some of the data will be manually entered (the same way that it would be manually recorded in a lab notebook), the advantage of DataJoint is that standard downstream processing steps can be run automatically on all new data with a single command. This is where the notion of a **data pipeline** comes into play. When the workflow of cleaning and processing the data, extracting important features, and performing basic analyses is all implemented in a DataJoint pipeline, minimal effort is required to analyze newly-collected data. Depending on the size of the raw files and the complexity of analysis, useful results may be available in a matter of minutes or hours. Because these results are stored in the database, they can be made available to anyone who is given access credentials for additional downstream analyses.

1.3.3 Won't the database get too big if all my data are there?

Typically, this is not a problem. If you find that your database is getting larger than a few dozen TB, DataJoint provides transparent solutions for storing very large chunks of data (larger than the 4 GB that can be natively stored as a LONGBLOB in MySQL). However, in many scenarios even long time series or images can be stored directly in the database with little effect on performance.

1.3.4 Why not just process the data and save them back to a file?

There are two main advantages to storing results in the database. The first is data integrity. Because the relationships between data are enforced by the structure of the database, DataJoint ensures that the metadata in the upstream nodes always correctly describes the computed results downstream in the pipeline. If a specific experimental session is deleted, for example, all the data extracted from that session are automatically removed as well, so there is no chance of "orphaned" data. Likewise, the database ensures that computations are atomic. This means that any computation performed on a dataset is performed in an all-or-none fashion. Either all of the data are processed and inserted, or none at all. This ensures that there are no incomplete data. Neither of these important features of data integrity can be guaranteed by a file system.

The second advantage of storing intermediate results in a data pipeline is flexible access. Accessing arbitrarily complex subsets of the data can be achieved with DataJoint's flexible query language. When data are stored in files, collecting the desired data requires trawling through the file hierarchy, finding and loading the files of interest, and selecting the interesting parts of the data.

This brings us to the final important question:

1.3.5 How do I get my data out?

This is the fun part. See *Queries* for details of the DataJoint query language directly from MATLAB and Python.

1.3.6 Interfaces

Multiple interfaces may be used to get the data into and out of the pipeline.

Some labs use third-party GUI applications such as HeidiSQL and Navicat, for example. These applications allow entering and editing data in tables similarly to spreadsheets.

The Helium Application (https://mattbdean.github.io/Helium/ and https://github.com/mattbdean/Helium) is web application for browsing DataJoint pipelines and entering new data. Matt Dean develops and maintains Helium under the direction of members of Karel Svoboda's lab at Janelia Research Campus and Vathes LLC.

Data may also be imported or synchronized into a DataJoint pipeline from exising LIMS (laboratory information management systems). For example, the International Brain Lab synchronizes data from an Alyx database. For implementation details, see https://github.com/int-brain-lab/IBL-pipeline.

Other labs (e.g. Sinz Lab) have developed GUI interfaces using the Flask web framework in Python.

GUIs, Google Docs integration, LIMS integration, Slack integration, *etc.*

1.4 Community

DataJoint was originally developed by working systems neuroscientists at Andreas Tolias' Lab at Baylor College of Medicine to meet the needs of their own research.

Below is a partial list of known DataJoint users. Please let us know if you would like to add another lab or make a correction.

1.4.1 Multi-lab collaboratives

1. International Brain Laboratory
2. Mesoscale Activity Project
3. IARPA MICrONS
4. Princeton U19 Project
5. UCSD U19 Project "Reverse Engineering the Brain Stem Circuits That Govern Exploratory Behavior"

1.4.2 Invidiual Labs

1. Tolias Lab (Andreas Tolias), Baylor College of Medicine
2. Siapas Lab (Athanassios G. Siapas), California Institute of Technology
3. Svoboda Lab (Karel Svoboda), Janelia Research Campus
4. Li Lab (Nuo Li), Baylor College of Medicine
5. Busse Lab (Laura Busse), Ludwig-Maximilians-Universität München, München, Germany
6. Katzner Lab (Steffen Katzner), Ludwig-Maximilians-Universität München, München, Germany

7. Sinz Lab (Fabian Sinz), Wilhelm Schickard Institute for Computer Science, Cyber Valley Initiative, University Tübingen
8. Berens Lab (Philipp Berens), Werner Reichardt Centre for Integrative Neuroscience, Tübingen, Germany
9. Euler Lab (Thomas Euler), Werner Reichardt Centre for Integrative Neuroscience, Tübingen, Germany
10. Bethge Lab (Matthias Bethge), Werner Reichardt Centre for Integrative Neuroscience, Tübingen, Germany
11. Shcheglovitov Lab (Alex Shcheglovitov) University of Utah
12. Moser Group (May-Britt Moser and Edvard Moser), Kavli Institute for Systems Neuroscience and Centre for Neural Computation, Norwegian University of Science and Technology (NTNU), Trondheim, Norway
13. Seung Lab (Sebastian Seung), Princeton University
14. Mouse Motor Lab (Mackenzie Mathis), Rowland Institute at Harvard University
15. Harvey Lab (Christopher Harvey), Harvard Medical School
16. Angelaki Lab (Dora Angelaki), New York University
17. Smirnakis Lab (Stelios Smirnakis), Harvard Medical School
18. McGinley Lab (Matthew McGinley), Baylor College of Medicine
19. Reimer Lab (Jacob Reimer), Baylor College of Medicine
20. Wang Lab (Fan Wang), Duke University
21. Applied Physics Laboratory, Johns Hopkins University

1.5 Publications

The following are some of the studies that used DataJoint for building their data pipelines.

1. Rosón, M. R., Bauer, Y., Kotkat, A. H., Berens, P., Euler, T., & Busse, L. (2019). Mouse dLGN receives functional input from a diverse population of retinal ganglion cells with limited convergence. Neuron, 102(2), 462-476.
2. Ecker, A. S., Sinz, F. H., Froudarakis, E., Fahey, P. G., Cadena, S. A., Walker, E. Y., … & Bethge, M. (2018). A rotation-equivariant convolutional neural network model of primary visual cortex. arXiv preprint arXiv:1809.10504.
3. Chettih, S. N., & Harvey, C. D. (2019). Single-neuron perturbations reveal feature-specific competition in V1. Nature, 567(7748), 334.
4. Denfield, G. H., Ecker, A. S., Shinn, T. J., Bethge, M., & Tolias, A. S. (2018). Attentional fluctuations induce shared variability in macaque primary visual cortex. Nature communications, 9(1), 2654.
5. Shan, Kevin Q., Evgueniy V. Lubenov, and Athanassios G. Siapas. "Model-based spike sorting with a mixture of drifting t-distributions." Journal of neuroscience methods 288 (2017): 82-98.
6. Reimer, J., McGinley, M. J., Liu, Y., Rodenkirch, C., Wang, Q., McCormick, D. A., & Tolias, A. S. (2016). Pupil fluctuations track rapid changes in adrenergic and cholinergic activity in cortex. Nature communications, 7, 13289.
7. Franke, K., Berens, P., Schubert, T., Bethge, M., Euler, T., & Baden, T. (2017). Inhibition decorrelates visual feature representations in the inner retina. Nature, 542(7642), 439.
8. Cadwell, Cathryn R., et al. "Electrophysiological, transcriptomic and morphologic profiling of single neurons using Patch-seq." Nature biotechnology 34.2 (2016): 199.

9. Shan, K. Q., Lubenov, E. V., Papadopoulou, M., & Siapas, A. G. (2016). Spatial tuning and brain state account for dorsal hippocampal CA1 activity in a non-spatial learning task. Elife, 5, e14321.
10. Jiang, X., Shen, S., Cadwell, C. R., Berens, P., Sinz, F., Ecker, A. S., … & Tolias, A. S. (2015). Principles of connectivity among morphologically defined cell types in adult neocortex. Science, 350(6264), aac9462.
11. Yatsenko, D., Josić, K., Ecker, A. S., Froudarakis, E., Cotton, R. J., & Tolias, A. S. (2015). Improved estimation and interpretation of correlations in neural circuits. PLoS computational biology, 11(3), e1004083.
12. Reimer, J., Froudarakis, E., Cadwell, C. R., Yatsenko, D., Denfield, G. H., & Tolias, A. S. (2014). Pupil fluctuations track fast switching of cortical states during quiet wakefulness. Neuron, 84(2), 355-362.
13. Erisken, S., Vaiceliunaite, A., Jurjut, O., Fiorini, M., Katzner, S., & Busse, L. (2014). Effects of locomotion extend throughout the mouse early visual system. Current Biology, 24(24), 2899-2907.
14. Froudarakis, E., Berens, P., Ecker, A. S., Cotton, R. J., Sinz, F. H., Yatsenko, D., … & Tolias, A. S. (2014). Population code in mouse V1 facilitates readout of natural scenes through increased sparseness. Nature neuroscience, 17(6), 851.
15. Ecker, A. S., Berens, P., Cotton, R. J., Subramaniyan, M., Denfield, G. H., Cadwell, C. R., … & Tolias, A. S (2014). State dependence of noise correlations in macaque primary visual cortex. Neuron, 82(1), 235-248.
16. Cotton, R. J., Froudarakis, E., Storer, P., Saggau, P., & Tolias, A. S. (2013). Three-dimensional mapping of microcircuit correlation structure. Frontiers in neural circuits, 7, 151.
17. Vaiceliunaite, A., Erisken, S., Franzen, F., Katzner, S., & Busse, L. (2013). Spatial integration in mouse primary visual cortex. Journal of neurophysiology, 110(4), 964-972.

. progress: 2.0 100% Dimitri

1.6 Publishing Data

DataJoint is a framework for building data pipelines that support rigorous flow of structured data between experimenters, data scientists, and computing agents *during* data acquisition and processing within a centralized project. Publishing final datasets for the outside world may require additional steps and conversion.

1.6.1 Provide access to a DataJoint server

One approach for publishing data is to grant public access to an existing pipeline. Then public users will be able to query the data pipelines using DataJoint's query language and output interfaces just like any other users of the pipeline. For security, this may require synchronizing the data onto a separate read-only public server.

1.6.2 Containerizing as a DataJoint pipeline

Containerization platforms such as docker allow convenient distribution of environments including database services and data. It is convenient to publish DataJoint pipelines as a docker container that deploys the populated DataJoint pipeline. One example of publishing a DataJoint pipeline as a docker container is > Sinz, F., Ecker, A.S., Fahey, P., Walker, E., Cobos, E., Froudarakis, E., Yatsenko, D., Pitkow, Z., Reimer, J. and Tolias, A., 2018. Stimulus domain transfer in recurrent models for large scale cortical population prediction on video. In Advances in Neural Information Processing Systems (pp. 7198-7209). https://www.biorxiv.org/content/early/2018/10/25/452672

The code and the data can be found at https://github.com/sinzlab/Sinz2018_NIPS

1.6.3 Exporting into a collection of files

Another option for publishing and archiving data is to export the data from the DataJoint pipeline into a collection of files. DataJoint provides features for exporting and importing sections of the pipeline. Several ongoing projects are implementing the capability to export from DataJoint pipelines into Neurodata Without Borders files.

1.7 Progress

Dimitri Yatsenko began development of DataJoint in Andreas S.Tolias' lab in the Neuroscience Department at Baylor College of Medicine in the fall of 2009. Initially implemented as a thin MySQL API in MATLAB, it defined the major principles of the DataJoint model.

Many students and postdocs in the lab as well as collaborators and early adopters have contributed to the project. Jacob Reimer and Emmanouil Froudarakis became early adopters in Andreas Tolias' Lab and propelled development. Alexander S. Ecker, Philipp Berens, Andreas Hoenselaar, and R. James Cotton contributed to the formulation of the overall requirements for the data model and critical reviews of DataJoint development.

Outside the Tolias lab, the first labs to adopt DataJoint (approx. 2010) were the labs of Athanassios G. Siapas at CalTech, Laura Busse and Steffen Katzner at the University of Tübingen.

In 2015, the Python implementation gained momentum with Edgar Y. Walker and Fabian Sinz joining as principal contributors.

In 2016, Andreas Tolias Lab joined the MICrONS project, using DataJoint to process volumes of neurophysiology and neuroanatomical data shared across large teams.

In 2016, Vathes LLC was founded to provide support to groups using DataJoint.

In 2017, DARPA awarded a small-business innovation research grant to Vathes LLC (Contract D17PC00162) to further develop and publicize the DataJoint framework.

In June 2018, the Princeton Neuroscience Institute, under the leadership of Prof. Carlos Brody, began funding a project to generate a detailed DataJoint user manual.

1.8 License

This documentation is distributed under the Creative Commons Attribution-ShareAlike 4.0 International Public License. CC BY-SA 4.0 with copyright to "DataJoint Contributors" and required URL reference to https://docs.datajoint.io.

1.9 Issues

This documentation is hosted in the GitHub repository https://github.com/datajoint/datajoint-docs.

For questions, issues, and requests concerning this documentation, please submit issues to the issue tracker, and they will be handled in order of their priority.

For questions and issues pertaining specifically to the client libraries, please use the corresponding issue trackers for datajoint-python and datajoint-matlab.

For general *How do I...* types of questions, please use StackExchange and tag your questions with `datajoint`.

Also see *Contribute*.

1.10 Contribute

This documentation is published on the DataJoint website. The documentation may be distributed under the terms of the *license* with a required reference to https://docs.datajoint.io and copyright to *DataJoint Contributors*.

The master source for this documentation is hosted on GitHub at https://github.com/datajoint/datajoint-docs. To report an issue with the documentation, please use the issue tracker.

To contribute, fork the documentation repository into a personal GitHub repository. Upon completing the contribution, please issue a pull request for review by the core DataJoint contributors. The documentation must be written in RestrcturedText using Sphinx conventions The README file of the repository provides instructions for building the documentation as well as writing style guidelines.

1.11 FAQs

1.11.1 How do I use GUIs with DataJoint?

It is common to enter data during experiments using a graphical user interface.

1.11.2 Does DataJoint support other programming languages?

DataJoint was originally developed to support MATLAB, followed by Python. DataJoint's data model and data representation are largely language independent, which means that any language with a DataJoint client can work with a data pipeline defined in any other language. DataJoint clients for other programming languages will be implemented based on demand. All languages must comply to the same data model and computation approach as defined in DataJoint: a simpler relational data model.

1.11.3 Is DataJoint another ORM?

Programmers are familiar with object-relational mappings (ORM) in various programming languages. Python in particular has several popular ORMs such as SQLAlchemy and Django ORM. The purpose of ORMs is to allow representations and manipulations of objects from the host programming language as data in a relational database. ORMs allow making objects persistent between program executions. ORMs create a bridge or a **mapping** between the object model used by the host language and the relational model allowed by the database. The result is always a compromise, usually toward the object model. ORMs usually forgo key concepts, features, and capabilities of the relational model for the sake of convenient programming constructs in the language.

In contrast, DataJoint implements a data model that is a refinement of the relational data model and that adheres to it faithfully without compromising its principles. DataJoint supports data integrity (entity integrity, referential integrity, and group integrity) and provides a fully capable relational query language. DataJoint remains absolutely data-centric, with the primary focus on the structure and integrity of the data pipeline. Other ORMs are more application-centric, primarily focusing on the application design while the database plays a secondary role supporting the application with object persistence and sharing.

1.11.4 How can I use DataJoint with a LIMS?

Lab Information Management Systems (LIMS)

1.11.5 What is the difference between DataJoint and Alyx?

Alyx is an experiment management database application developed in Kenneth Harris' lab at UCL.

Alyx is an application with a fixed pipeline design with a nice graphical user interface. In contrast, DataJoint is a general-purpose library for designing and building data processing pipelines.

Alyx is geared towards ease of data entry and tracking for a specific workflow (e.g. mouse colony information and some pre-specified experiments) and data types. DataJoint could be used as a more general purposes tool to design, implement, and execute processing on such workflows/pipelines from scratch, and DataJoint focuses on flexibility, data integrity, and ease of data analysis. The purposes are partly overlapping and complementary. The International Brain Lab project is developing a bridge from Alyx to DataJoint, hosted as an open-source project. It implements a DataJoint schema that replicates the major features of the Alyx application and a synchronization script from an existing Alyx database to its DataJoint counterpart.

1.12 Release Notes

1.12.1 0.12.8 – Jan 12, 2021

- table.children, .parents, .descendents, and ancestors can return queryable objects. PR #833
- Load dependencies before querying dependencies. (#179) PR #833
- Fix display of part tables in *schema.save*. (#821) PR #833
- Add *schema.list_tables*. (#838) PR #844
- Fix minio new version regression. PR #847
- Add more S3 logging for debugging. (#831) PR #832
- Convert testing framework from TravisCI to GitHub Actions (#841) PR #840

1.12.2 0.12.7 – Oct 27, 2020

- Fix case sensitivity issues to adapt to MySQL 8+. PR #819
- Fix pymysql regression bug (#814) PR #816
- Adapted attribute types now have *dtype=object* in all recarray results. PR #811

1.12.3 0.12.6 – May 15, 2020

- Add *order_by* to *dj.kill* (#668, #779) PR #775, #783
- Add explicit S3 bucket and file storage location existence checks (#748) PR #781
- Modify *_update* to allow nullable updates for strings/date (#664) PR #760
- Avoid logging events on auxiliary tables (#737) PR #753
- Add *kill_quick* and expand display to include host (#740) PR #741
- Bugfix - pandas insert fails due to additional *index* field (#666) PR #776
- Bugfix - *delete_external_files=True* does not remove from S3 (#686) PR #781
- Bugfix - pandas fetch throws error when *fetch_format='frame'* PR #774

1.12.4 0.12.5 – Feb 24, 2020

- Rename module *dj.schema* into *dj.schemas*. *dj.schema* remains an alias for class *dj.Schema*. (#731) PR #732
- *dj.create_virtual_module* is now called *dj.VirtualModule* (#731) PR #732
- Bugfix - SSL *KeyError* on failed connection (#716) PR #725
- Bugfix - Unable to run unit tests using nosetests (#723) PR #724
- Bugfix - *suppress_errors* does not suppress loss of connection error (#720) PR #721

1.12.5 0.12.4 – Jan 14, 2020

- Support for simple scalar datatypes in blobs (#690) PR #709
- Add support for the *serial* data type in declarations: alias for *bigint unsigned auto_increment* PR #713
- Improve the log table to avoid primary key collisions PR #713
- Improve documentation in README PR #713

1.12.6 0.12.3 – Nov 22, 2019

- Bugfix - networkx 2.4 causes error in diagrams (#675) PR #705
- Bugfix - include table definition in doc string and help (#698, #699) PR #706
- Bugfix - job reservation fails when native python datatype support is disabled (#701) PR #702

1.12.7 0.12.2 – Nov 11, 2019

- Bugfix - Convoluted error thrown if there is a reference to a non-existent table attribute (#691) PR #696
- Bugfix - Insert into external does not trim leading slash if defined in *dj.config['stores']['<store>']['location']* (#692) PR #693

1.12.8 0.12.1 – Nov 2, 2019

- Bugfix - AttributeAdapter converts into a string (#684) PR #688

1.12.9 0.12.0 – Oct 31, 2019

- Dropped support for Python 3.4
- Support secure connections with TLS (aka SSL) PR #620
- Convert numpy array from python object to appropriate data type if all elements are of the same type (#587) PR #608
- Remove expression requirement to have additional attributes (#604) PR #604
- Support for filepath datatype (#481) PR #603, #659
- Support file attachment datatype (#480, #592, #637) PR #659
- Fetch return a dict array when specifying *as_dict=True* for specified attributes. (#595) PR #593
- Support of ellipsis in *proj*: *query_expression.proj(.., '-movie')* (#499) PR #578
- Expand support of blob serialization (#572, #520, #427, #392, #244, #594) PR #577
- Support for alter (#110) PR #573
- Support for *conda install datajoint* via *conda-forge* channel (#293)
- *dj.conn()* accepts a *port* keyword argument (#563) PR #571
- Support for UUID datatype (#562) PR #567
- *query_expr.fetch("KEY", as_dict=False)* returns results as `np.recarray`(#414) PR #574
- *dj.ERD* is now called *dj.Diagram* (#255, #546) PR #565
- *dj.Diagram* underlines "distinguished" classes (#378) PR #557
- Accept alias for supported MySQL datatypes (#544) PR #545
- Support for pandas in *fetch* (#459, #537) PR #534
- Support for ordering by "KEY" in *fetch* (#541) PR #534
- Add config to enable python native blobs PR #672, #676
- Add secure option for external storage (#663) PR #674, #676
- Add blob migration utility from DJ011 to DJ012 PR #673
- Improved external storage - a migration script needed from version 0.11 (#467, #475, #480, #497) PR #532
- Increase default display rows (#523) PR #526
- Bugfixes (#521, #205, #279, #477, #570, #581, #597, #596, #618, #633, #643, #644, #647, #648, #650, #656)
- Minor improvements (#538)

1.12.10 0.11.1 – Nov 15, 2018

- Fix ordering of attributes in proj (#483 and #516)
- Prohibit direct insert into auto-populated tables (#511)

1.12.11 0.11.0 – Oct 25, 2018

- Full support of dependencies with renamed attributes using projection syntax (#300, #345, #436, #506, #507)
- Rename internal class and module names to comply with terminology in documentation (#494, #500)
- Full support of secondary indexes (#498, 500)
- ERD no longer shows numbers in nodes corresponding to derived dependencies (#478, #500)
- Full support of unique and nullable dependencies (#254, #301, #493, #495, #500)
- Improve memory management in `populate` (#461, #486)
- Fix query errors and redundancies (#456, #463, #482)

1.12.12 0.10.1 – Aug 28, 2018

- Fix ERD Tooltip message (#431)
- Networkx 2.0 support (#443)
- Fix insert from query with skip_duplicates=True (#451)
- Sped up queries (#458)
- Bugfix in restriction of the form (A & B) * B (#463)
- Improved error messages (#466)

1.12.13 0.10.0 – Jan 10, 2018

- Deletes are more efficient (#424)
- ERD shows table definition on tooltip hover in Jupyter (#422)
- S3 external storage
- Garbage collection for external sorage
- Most operators and methods of tables can be invoked as class methods rather than instance methods (#407)
- The schema decorator object no longer requires locals() to specify the context
- Compatibility with pymysql 0.8.0+
- More efficient loading of dependencies (#403)

1.12.14 0.9.0 – Nov 17, 2017

- Made graphviz installation optional
- Implement file-based external storage
- Implement union operator +
- Implement file-based external storage

1.12.15 0.8.0 – Jul 26, 2017

Documentation and tutorials available at https://docs.datajoint.io and https://tutorials.datajoint.io * improved the ERD graphics and features using the graphviz libraries (#207, #333) * improved password handling logic (#322, #321) * the use of the `contents` property to populate tables now only works in `dj.Lookup` classes (#310). * allow suppressing the display of size of query results through the `show_tuple_count` configuration option (#309) * implemented renamed foreign keys to spec (#333) * added the `limit` keyword argument to populate (#329) * reduced the number of displayed messages (#308) * added `size_on_disk` property for dj.Schema() objects (#323) * job keys are entered in the jobs table (#316, #243) * simplified the `fetch` and `fetch1` syntax, deprecating the `fetch[...]` syntax (#319) * the jobs tables now store the connection ids to allow identifying abandoned jobs (#288, #317)

1.12.16 0.5.0 (#298) – Mar 8, 2017

- All fetched integers are now 64-bit long and all fetched floats are double precision.
- Added `dj.create_virtual_module`

1.12.17 0.4.10 (#286) – Feb 6, 2017

- Removed Vagrant and Readthedocs support
- Explicit saving of configuration (issue #284)

1.12.18 0.4.9 (#285) – Feb 2, 2017

- Fixed setup.py for pip install

1.12.19 0.4.7 (#281) – Jan 24, 2017

- Fixed issues related to order of attributes in projection.

1.12.20 0.4.6 (#277) – Dec 22, 2016

- Proper handling of interruptions during populate

1.12.21 0.4.5 (#274) – Dec 20, 2016

- Populate reports how many keys remain to be populated at the start.

1.12.22 0.4.3 (#271) – Dec 6, 2016

- Fixed aggregation issues (#270)
- datajoint no longer attempts to connect to server at import time
- dropped support of view (reversed #257)
- more elegant handling of insufficient privileges (#268)

1.12.23 0.4.2 (#267) – Dec 6, 2016

- improved table appearance in Jupyter

1.12.24 0.4.1 (#266) – Oct 28, 2016

- bugfix for very long error messages

1.12.25 0.3.9 – Sep 27, 2016

- Added support for datatype `YEAR`
- Fixed issues with `dj.U` and the `aggr` operator (#246, #247)

1.12.26 0.3.8 – Aug 2, 2016

- added the `_update` method in `base_relation`. It allows updating values in existing tuples.
- bugfix in reading values of type double. Previously it was cast as float32.

1.12.27 0.3.7 – Jul 31, 2016

- added parameter `ignore_extra_fields` in `insert`
- `insert(..., skip_duplicates=True)` now relies on `SELECT IGNORE`. Previously it explicitly checked if tuple already exists.
- table previews now include blob attributes displaying the string <BLOB>

1.12.28 0.3.6 – Jul 30, 2016

- bugfix in `schema.spawn_missing_classes`. Previously, spawned part classes would not show in ERDs.
- dj.key now causes fetch to return as a list of dicts. Previously it was a recarray.

1.12.29 0.3.5

- `dj.set_password()` now asks for user confirmation before changing the password.
- fixed issue #228

1.12.30 0.3.4

- Added method the `ERD.add_parts` method, which adds the part tables of all tables currently in the ERD.
- `ERD() + arg` and `ERD() - arg` can now accept relation classes as arg.

1.12.31 0.3.3

- Suppressed warnings (redirected them to logging). Previoiusly, scipy would throw warnings in ERD, for example.
- Added ERD.from_sequence as a shortcut to combining the ERDs of multiple sources
- ERD() no longer text the context argument.
- ERD.draw() now takes an optional context argument. By default uses the caller's locals.

1.12.32 0.3.2

- Fixed issue #223: `insert` can insert relations without fetching.
- ERD() now takes the `context` argument, which specifies in which context to look for classes. The default is taken from the argument (schema or relation).
- ERD.draw() no longer has the `prefix` argument: class names are shown as found in the context.

CHAPTER

TWO

SERVER ADMINISTRATION

2.1 Database Server Hosting

Let's say a person, a lab, or a multi-lab consortium decide to use DataJoint as their data pipeline platform. What IT resources and support will be required?

DataJoint uses a MySQL-compatible database server such as MySQL, MariaDB, Percona Server, or Amazon Aurora to store the structured data used for all relational operations. Large blocks of data associated with these records such as multidimensional numeric arrays (signals, images, scans, movies, etc) can be stored within the database or stored in additionally configured *bulk storage*.

The first decisions you need to make are where this server will be hosted and how it will be administered. The server may be hosted on your personal computer, on a dedicated machine in your lab, or in a cloud-based database service.

2.1.1 Cloud hosting

Increasingly, many teams make use of cloud-hosted database services, which allow great flexibility and easy administration of the database server. A cloud hosting option will be provided through https://hub.datajoint.io. The hub simplifies the setup for labs that wish to host their data pipelines in the cloud and allows sharing pipelines between multiple groups and locations. Being an open-source solution, other cloud services such as Amazon RDS can also be used in this role, albeit with less DataJoint-centric customization.

2.1.2 Self hosting

In the most basic configuration, the relational database software and DataJoint are installed onto a single computer which is used by an individual user. To support a small group of users, a larger computer can be used instead and configured for remote access. As the number of users grows, individual workstations can be installed with the DataJoint software and used to connect to a larger and more specialized centrally located database server machine.

For even larger groups or multi-site collaborations, multiple database servers may be configured in a replicated fashion to support larger workloads and simultaneous multi-site access. The following section provides some basic guidelines for these configurations here and in the subsequent sections of the documentation.

2.1.3 General server / hardware support requirements

The following table lists some likely scenarios for DataJoint database server deployments and some reasonable estimates of the required computer hardware. The required IT/systems support needed to ensure smooth operations in the absence of local database expertise is also listed.

Table 1: IT infrastructures

Usage Scenario	DataJoint Database Computer	Required IT Support
Single User	Personal Laptop or Workstation	Self-Supported or Ad-Hoc General IT Support
Small Group (e.g. 2-10 Users)	Workstation or Small Server	Ad-Hoc General or Experienced IT Support
Medium Group (e.g. 10-30 Users)	Small to Medium Server	Ad-Hoc/Part Time Experienced or Specialized IT Support
Large Group/Department (e.g. 30-50+ Users)	Medium/Large Server or Multi-Server Replication	Part Time/Dedicated Experienced or Specialized IT Support
Multi-Location Collaboration (30+ users, Geographically Distributed)	Large Server, Advanced Replication	Dedicated Specialized IT Support

2.2 Relational Database Server

2.2.1 Hardware considerations

As in any computer system, CPU, RAM memory, disk storage, and network speed are important components of performance. The relational database component of DataJoint is no exception to this rule. This section discusses the various factors relating to selecting a server for your DataJoint pipelines.

CPU

CPU speed and parallelism (number of cores/threads) will impact the speed of queries and the number of simultaneous queries which can be efficiently supported by the system. It is a good rule of thumb to have enough cores to support the number of active users and background tasks you expect to have running during a typical 'busy' day of usage. For example, a team of 10 people might want to have 8 cores to support a few active queries and background tasks.

RAM

The amount of RAM will impact the amount of DataJoint data kept in memory, allowing for faster querying of data since the data can be searched and returned to the user without needing to access the slower disk drives. It is a good idea to get enough memory to fully store the more important and frequently accessed portions of your dataset with room to spare, especially if in-database blob storage is used instead of external *bulk storage*.

Disk

The disk storage for a DataJoint database server should have fast random access, ideally with flash-based storage to eliminate the rotational delay of mechanical hard drives.

Networking

When network connections are used, network speed and latency are important to ensure that large query results can be quickly transferred across the network and that delays due to data entry/query round-trip have minimal impact on the runtime of the program.

General recommendations

DataJoint datasets can consist of many thousands or even millions of records. Generally speaking one would want to make sure that the relational database system has sufficient CPU speed and parallelism to support a typical number of concurrent users and to execute searches quickly. The system should have enough RAM to store the primary key values of commonly used tables and operating system caches. Disk storage should be fast enough to support quick loading of and searching through the data. Lastly, network bandwidth must be sufficient to support transferring user records quickly.

2.2.2 Large-scale installations

Database replication may be beneficial if system downtime or precise database responsiveness is a concern Replication can allow for easier coordination of maintenance activities, faster recovery in the event of system problems, and distribution of the database workload across server machines to increase throughput and responsiveness.

Master-slave replication

Master/slave replication allows for creation of a read-only database copy which is updated in real time. This copy can be used for backup or queries which are not time sensitive. It can also be upgraded with read-write usage in the event that the main database fails.

Multi-master replication

Multi-master replication configurations allow for all replicas to be used in a read/write fashion, with the workload being distributed among all machines. However, multi-master replication is also more complicated, requiring front-end machines to distribute the workload, similar performance characteristics on all replicas to prevent bottlenecks, and redundant network connections to ensure the replicated machines are always in sync.

2.2.3 Recommendations

It is usually best to go with the simplest solution which can suit the requirements of the installation, adjusting workloads where possible and adding complexity only as needs dictate.

Resource requirements of course depend on the data collection and processing needs of the given pipeline, but there are general size guidelines that can inform any system configuration decisions. A reasonably powerful workstation or small server should support the needs of a small group (2-10 users). A medium or large server should support the needs of a larger user community (10-30 users). A replicated or distributed setup of 2 or more medium or large servers may be required in larger cases. These requirements can be reduced through the use of external or cloud storage, which is discussed in the subsequent section.

Table 2: Recommendations

Usage Scenario	DataJoint Database Computer	Hardware Recommendation
Single User	Personal Laptop or Workstation	4 Cores, 8-16GB or more of RAM, SSD or better storage
Small Group (e.g. 2-10 Users)	Workstation or Small Server	8 or more Cores, 16GB or more of RAM, SSD or better storage
Medium Group (e.g. 10-30 Users)	Small to Medium Server	8-16 or more Cores, 32GB or more of RAM, SSD/RAID or better storage
Large Group/Department (e.g. 30-50+ Users)	Medium/Large Server or Multi-Server Replication	16-32 or more Cores, 64GB or more of RAM, SSD Raid storage, multiple machines
Multi-Location Collaboration (30+ users, Geographically Distributed)	Large Server, Advanced Replication	16-32 or more Cores, 64GB or more of RAM, SSD Raid storage, multiple machines; potentially multiple machines in multiple locations

2.2.4 Docker

A Docker image is available for a MySQL server configured to work with DataJoint: https://github.com/datajoint/mysql-docker.

2.3 User Management

Create user accounts on the MySQL server. For example, if your username is alice, the SQL code for this step is:

```sql
CREATE USER 'alice'@'%' IDENTIFIED BY 'alices-secret-password';
```

Existing users can be listed using the following SQL:

```sql
SELECT user, host from mysql.user;
```

Teams that use DataJoint typically divide their data into schemas grouped together by common prefixes. For example, a lab may have a collection of schemas that begin with `common_`. Some common processing may be organized into several schemas that begin with `pipeline_`. Typically each user has all privileges to schemas that begin with her username.

For example, alice may have privileges to select and insert data from the common schemas (but not create new tables), and have all privileges to the pipeline schemas.

Then the SQL code to grant her privileges might look like:

```sql
GRANT SELECT, INSERT ON `common\_%`.* TO 'alice'@'%';
GRANT ALL PRIVILEGES ON `pipeline\_%`.* TO 'alice'@'%';
GRANT ALL PRIVILEGES ON `alice\_%`.* TO 'alice'@'%';
```

To note, the `ALL PRIVILEGES` option allows the user to create and remove databases without administrator intervention.

Once created, a user's privileges can be listed using the `SHOW GRANTS` statement.

```sql
SHOW GRANTS FOR 'alice'@'%';
```

2.3.1 Grouping with Wildcards

Depending on the complexity of your installation, using additional wildcards to group access rules together might make managing user access rules simpler. For example, the following equivalent convention:

```
GRANT ALL PRIVILEGES ON `user_alice\_%`.* TO 'alice'@'%';
```

Could then facilitate using a rule like:

```
GRANT SELECT ON `user\_%\_%`.* TO 'bob'@'%';
```

to enable `bob` to query all other users tables using the `user_username_database` convention without needing to explicitly give him access to `alice\_%`, `charlie\_%`, and so on.

This convention can be further expanded to create notions of groups and protected schemas for background proccesing, etc. For example:

```
GRANT ALL PRIVILEGES ON `group\_shared\_%`.* TO 'alice'@'%';
GRANT ALL PRIVILEGES ON `group\_shared\_%`.* TO 'bob'@'%';

GRANT ALL PRIVILEGES ON `group\_wonderland\_%`.* TO 'alice'@'%';
GRANT SELECT ON `group\_wonderland\_%`.* TO 'alice'@'%';
```

could allow both bob an alice to read/write into the `group\_shared` databases, but in the case of the `group\_wonderland` databases, read write access is restricted to alice.

2.4 Bulk Storage Systems

2.4.1 Why External Bulk Storage?

DataJoint supports the storage of large data objects associated with relational records externally from the MySQL Database itself. This is significant and useful for a number of reasons.

Cost

One of these is that the high-performance storage commonly used in database systems is more expensive than that used in more typical commodity storage, and so storing the smaller identifying information typically used in queries on fast, relational database storage and storing the larger bulk data used for analysis or processing on lower cost commodity storage can allow for large savings in storage expense.

Flexibility

Storing bulk data separately also facilitates more flexibility in usage, since the bulk data can managed using separate maintenance processes than that in the relational storage.

For example, larger relational databases may require many hours to be restored in the event of system failures. If the relational portion of the data is stored separately, with the larger bulk data stored on another storage system, this downtime can be reduced to a matter of minutes. Similarly, due to the lower cost of bulk commodity storage, more emphasis can be put into redundancy of this data and backups to help protect the non-relational data.

Performance

Storing the non-relational bulk data separately can have system performance impacts by removing data transfer, disk I/O, and memory load from the database server and shifting these to the bulk storage system. Additionally, datajoint supports caching of bulk data records which can allow for faster processing of records which already have been retrieved in previous queries.

Data Sharing

DataJoint provides pluggable support for different external bulk storage backends, which can provide benefits for data sharing by publishing bulk data to S3-Protocol compatible data shares both in the cloud and on locally managed systems and other common tools for data sharing, such as Globus, etc.

2.4.2 Bulk Storage Scenarios

Typical bulk storage considerations relate to the cost of the storage backend per unit of storage, the amount of data which will be stored, the desired focus of the shared data (system performance, data flexibility, data sharing), and data access. Some common scenarios are given in the following table:

Scenario	Storage Solution	System Requirements	Notes
Local Object Cache	Local External Storage	Local Hard Drive	Used to Speed Access to other Storage
LAN Object Cache	Network External Storage	Local Network Share	Used to Speed Access to other storage, reduce Cloud/Network Costs/Overhead
Local Object Store	Local/Network External Storage	Local/Network Storage	Used to store objects externally from the database
Local S3-Compatible Store	Local S3-Compatible Server	Network S3-Server	Used to host S3-Compatible services locally (e.g. minio) for internal use or to lower cloud costs
Cloud S3-Compatible Storage	Cloud Provider	Internet Connectivity	Used to reduce/remove requirement for external storage management, data sharing
Globus Storage	Globus Endpoint	Local/Local Network Storage, Internet Connectivity	Used for institutional data transfer or publishing.

2.4.3 Bulk Storage Considerations

Although external bulk storage provides a variety of advantages for storage cost and data sharing, it also uses slightly different data input/retrieval semantics and as such has different performance characteristics.

Performance Characteristics

In the direct database connection scenario, entire result sets are either added or retrieved from the database in a single stream action. In the case of external storage, individual record components are retrieved in a set of sequential actions per record, each one subject to the network round trip to the given storage medium. As such, tables using many small records may be ill suited to external storage usage in the absence of a caching mechanism. While some of these impacts may be addressed by code changes in a future release of DataJoint, to some extent, the impact is directly related from needing to coordinate the activities of the database data stream with the external storage system, and so cannot be avoided.

Network Traffic

Some of the external storage solutions mentioned above incur cost both at a data volume and transfer bandwidth level. The number of users querying the database, data access, and use of caches should be considered in these cases to reduce this cost if applicable.

Data Coherency

When storing all data directly in the relational data store, it is relatively easy to ensure that all data in the database is consistent in the event of system issues such as crash recoveries, since MySQL's relational storage engine manages this for you. When using external storage however, it is important to ensure that any data recoveries of the database system are paired with a matching point-in-time of the external storage system. While DataJoint does use hashing to help facilitate a guarantee that external files are uniquely named throughout their lifecycle, the pairing of a given relational dataset against a given filesystem state is loosely coupled, and so an incorrect pairing could result in processing failures or other issues.

2.5 External Store

DataJoint organizes most of its data in a relational database. Relational databases excel at representing relationships between entities and storing structured data. However, relational databases are not particularly well-suited for storing large continuous chunks of data such as images, signals, and movies. An attribute of type `longblob` can contain an object up to 4 GiB in size (after compression) but storing many such large objects may hamper the performance of queries on the entire table. A good rule of thumb is that objects over 10 MiB in size should not be put in the relational database. In addition, storing data in cloud-hosted relational databases (e.g. AWS RDS) may be more expensive than in cloud-hosted simple storage systems (e.g. AWS S3).

DataJoint allows the use of *external* storage to store large data objects within its relational framework but outside of the main database.

Defining an externally-stored attribute is used using the notation `blob@storename` (see also: *definition syntax*) and works the same way as a `longblob` attribute from the users perspective. However, its data are stored in an external storage system rather than in the relational database.

Various systems can play the role of external storage, including a shared file system accessible to all team members with access to these objects or a cloud storage solutions such as AWS S3.

For example, the following table stores motion-aligned two-photon movies.

```
# Motion aligned movies
-> twophoton.Scan
---
aligned_movie :  blob@external  # motion-aligned movie in 'external' store
```

All *insert* and *fetch* operations work identically for *external* attributes as they do for *blob* attributes, with the same serialization protocol. Similar to *blobs*, *external* attributes cannot be used in restriction conditions.

Multiple external storage configurations may be used simultaneously with the `@storename` portion of the attribute definition determining the storage location.

```
# Motion aligned movies
-> twophoton.Scan
---
aligned_movie :  blob@external-raw  # motion-aligned movie in 'external-raw' store
```

2.5.1 Principles of operation

External storage is organized to emulate individual attribute values in the relational database. DataJoint organizes external storage to preserve the same data integrity principles as in relational storage.

1. The external storage locations are specified in the DataJoint connection configuration with one specification for each store.

```
dj.config['stores'] = {
  'external': dict(  # 'regular' external storage for this pipeline
                protocol='s3',
                endpoint='s3.amazonaws.com:9000',
                bucket = 'testbucket',
                location = 'datajoint-projects/lab1',
                access_key='1234567',
                secret_key='foaf1234'),
  'external-raw': dict( # 'raw' storage for this pipeline
                protocol='file',
                location='/net/djblobs/myschema')
}
# external object cache - see fetch operation below for details.
dj.config['cache'] = '/net/djcache'
```

2. Each schema corresponds to a dedicated folder at the storage location with the same name as the database schema.

3. Stored objects are identified by the SHA-256 hashes (in web-safe base-64 ASCII) of their serialized contents. This scheme allows for the same object—used multiple times in the same schema—to be stored only once.

4. In the `external-raw` storage, the objects are saved as files with the hash as the filename.

5. In the `external` storage, external files are stored in a directory layout corresponding to the hash of the filename. By default, this corresponds to the first 2 characters of the hash, followed by the second 2 characters of the hash, followed by the actual file.

6. Each database schema has an auxiliary table named `~external_<storename>` for each configured external store.

   It is automatically created the first time external storage is used. The primary key of `~external_<storename>` is the hash of the data (for blobs and attachments) or of the relative paths to the files for filepath-based storage. Other attributes are the `count` of references by tables in the schema, the `size` of the object in bytes, and the `timestamp` of the last event (creation, update, or deletion).

   Below are sample entries in `~external_<storename>`.

Table 3: ~external_raw

HASH	size	filepath	contents_ha	timestamp
1GEqtEU6JY VZio1ga25v	1039536788	NULL	NULL	2017-06-07 23:14:01

The fields *filepath* and *contents_hash* relate to the *filepath* datatype, which will be discussed separately.

7. Attributes of type `@<storename>` are declared as renamed *foreign keys* referencing the `~external_<storename>` table (but are not shown as such to the user).
8. The *insert* operation encodes and hashes the blob data. If an external object is not present in storage for the same hash, the object is saved and if the save operation is successful, corresponding entities in table `~external_<storename>` for that store are created.
9. The *delete* operation first deletes the foreign key reference in the target table. The external table entry and actual external object is not actually deleted at this time (*soft-delete*).
10. The *fetch* operation uses the hash values to find the data.

    In order to prevent excessive network overhead, a special external store named `cache` can be configured. If the `cache` is enabled, the `fetch` operation need not access `~external_<storename>` directly. Instead `fetch` will retrieve the cached object without downloading directly from the *real* external store.
11. Cleanup is performed regularly when the database is in light use or off-line.
12. DataJoint never removes objects from the local `cache` folder. The `cache` folder may just be periodically emptied entirely or based on file access date. If dedicated `cache` folders are maintained for each schema, then a special procedure will be provided to remove all objects that are no longer listed in `~external_<storename>`.

Data removal from external storage is separated from the delete operations to ensure that data are not lost in race conditions between inserts and deletes of the same objects, especially in cases of transactional processing or in processes that are likely to get terminated. The cleanup steps are performed in a separate process when the risks of race conditions are minimal. The process performing the cleanups must be isolated to prevent interruptions resulting in loss of data integrity.

2.5.2 Configuration

The following steps must be performed to enable external storage:

1. Assign external location settings for each storage as shown in the *Step 1* example above.

   Use `dj.config` for configuration.

   - `protocol` [*s3*, *file*] Specifies whether *s3* or *file* external storage is desired.
   - `endpoint` [*s3*] Specifies the remote endpoint to the external data for all schemas as well as the target port.
   - `bucket` [*s3*] Specifies the appropriate *s3* bucket organization.
   - `location` [*s3*, *file*] Specifies the subdirectory within the root or bucket of store to preserve data. External objects are thus stored remotely with the following path structure: `<bucket (if applicable)>/<location>/<schema_name>/<subfolding_strategy>/<object>`.
   - `access_key` [*s3*] Specifies the access key credentials for accessing the external location.
   - `secret_key` [*s3*] Specifies the secret key credentials for accessing the external location.

   - `secure` [*s3*] Optional specification to establish secure external storage connection with TLS (aka SSL, HTTPS). Defaults to `False`.

2. Optionally, for each schema specify the `cache` folder for local fetch cache.

   This is done by saving the path in the `cache` key of the DataJoint configuration dictionary:

   ```
   dj.config['cache'] = '/temp/dj-cache'
   ```

2.5.3 Cleanup

Deletion of records containing externally stored blobs is a *soft-delete* which only removes the database-side records from the database. To cleanup the external tracking table or the actual external files, a separate process is provided as follows.

To remove only the tracking entries in the external table, call `delete` on the `~external_<storename>` table for the external configuration with the argument `delete_external_files=False`.

Note: Currently, cleanup operations on a schema's external table are not 100% transaction safe and so must be run when there is no write activity occurring in tables which use a given schema / external store pairing.

```
>>> schema.external['external_raw'].delete(delete_external_files=False)
```

To remove the tracking entries as well as the underlying files, call *delete* on the external table for the external configuration with the argument *delete_external_files=True*.

```
>>> schema.external['external_raw'].delete(delete_external_files=True)
```

Note: Setting `delete_external_files=True` will always attempt to delete the underlying data file, and so should not typically be used with the `filepath` datatype.

2.5.4 Migration between DataJoint v0.11 and v0.12

Note: Please read carefully if you have used external storage in DataJoint v0.11!

The initial implementation of external storage was reworked for DataJoint v0.12. These changes are backward-incompatible with DataJoint v0.11 so care should be taken when upgrading. This section outlines some details of the change and a general process for upgrading to a format compatible with DataJoint v0.12 when a schema rebuild is not desired.

The primary changes to the external data implementation are:

- The external object tracking mechanism was modified. Tracking tables were extended for additional external datatypes and split into per-store tables to improve database performance in schemas with many external objects.
- The external storage format was modified to use a nested subfolder structure (*folding*) to improve performance and interoperability with some filesystems that have limitations or performance problems when storing large numbers of files in single directories.

Depending on the circumstances, the simplest way to migrate data to v0.12 may be to drop and repopulate the affected schemas. This will construct the schema and storage structure in the v0.12 format and save the need for database migration. When recreation is not possible or is not preferred to upgrade to DataJoint v0.12, the following process should be followed:

1) Stop write activity to all schemas using external storage.
2) Perform a full backup of your database(s).
3) Upgrade your DataJoint installation to v0.12
4) Adjust your external storage configuration (in *datajoint.config*) to the new v0.12 configuration format (see above).
5) Migrate external tracking tables for each schema to use the new format. For instance in Python:

```
    >>> import datajoint.migrate as migrate
    >>> db_schema_name='schema_1'
    >>> external_store='raw'
    >>> migrate.migrate_dj011_external_blob_storage_to_dj012(db_schema_name, external_store)

6) Verify pipeline functionality after this process has completed. For instance in Python:
```

```
>>> x = myschema.TableWithExternal.fetch('external_field', limit=1)[0]
```

Note: This migration function is provided on a best-effort basis, and will convert the external tracking tables into a format which is compatible with DataJoint v0.12. While we have attempted to ensure correctness of the process, all use-cases have not been heavily tested. Please be sure to fully back-up your data and be prepared to investigate problems with the migration, should they occur.

Please note:

- The migration only migrates the tracking table format and does not modify the backing file structure to support *folding*. The DataJoint v0.12 logic is able to work with this format, but to take advantage of the new backend storage, manual adjustment of the tracking table and files, or a full rebuild of the schema should be performed.
- Additional care to ensure all clients are using v0.12 should be taken after the upgrade. Legacy clients may incorrectly create data in the old format which would then need to be combined or otherwise reconciled with the data in v0.12 format. You might wish to take the opportunity to version-pin your installations so that future changes requiring controlled upgrades can be coordinated on a system wide basis.

2.6 Backups and Recovery

Backing up your DataJoint installation is critical to ensuring that your work is safe and can be continued in the event of system failures, and several mechanisms are available to use.

Much like your live installation, your backup will consist of two portions:

- Backup of the Relational Data
- Backup of optional external bulk storage

This section primarily deals with backup of the relational data since most of the optional bulk storage options use “regular” flat-files for storage and can be backed up via any “normal” disk backup regime.

There are many options to backup MySQL; subsequent sections discuss a few options.

2.6.1 Cloud hosted backups

In the case of cloud-hosted options, many cloud vendors provide automated backup of your data, and some facility for downloading such backups externally. Due to the wide variety of cloud-specific options, discussion of these options falls outside of the scope of this documentation. However, since the cloud server is also a MySQL server, other options listed here may work for your situation.

2.6.2 Disk-based backup

The simplest option for many cases is to perform a disk-level backup of your MySQL installation using standard disk backup tools. It should be noted that all database activity should be stopped for the duration of the backup to prevent errors with the backed up data. This can be done in one of two ways:

- Stopping the MySQL server program
- Using database locks

These methods are required since MySQL data operations can be ongoing in the background even when no user activity is ongoing. To use a database lock to perform a backup, the following commands can be used as the MySQL administrator:

```
FLUSH TABLES WITH READ LOCK;
UNLOCK TABLES;
```

The backup should be performed between the issuing of these two commands, ensuring the database data is consistent on disk when it is backed up.

2.6.3 MySQLDump

Disk based backups may not be feasible for every installation, or a database may require constant activity such that stopping it for backups is not feasible. In such cases, the simplest option is MySQLDump, a command line tool that prints the contents of your database contents in SQL form.

This tool is generally acceptable for most cases and is especially well suited for smaller installations due to its simplicity and ease of use.

For larger installations, the lower speed of MySQLDump can be a limitation, since it has to convert the database contents to and from SQL rather than dealing with the database files directly. Additionally, since backups are performed within a transaction, the backup will be valid up to the time the backup began rather than to its completion, which can make ensuring that the latest data are fully backed up more difficult as the time it takes to run a backup grows.

2.6.4 Percona XTraBackup

The Percona `xtrabackup` tool provides near-realtime backup capability of a MySQL installation, with extended support for replicated databases, and is a good tool for backing up larger databases.

However, this tool requires local disk access as well as reasonably fast backup media, since it builds an ongoing transaction log in real time to ensure that backups are valid up to the point of their completion. This strategy fails if it cannot keep up with the write speed of the database. Further, the backups it generates are in binary format and include incomplete database transactions, which require careful attention to detail when restoring.

As such, this solution is recommended only for advanced use cases or larger databases where limitations of the other solutions may apply.

2.6.5 Locking and DDL issues

One important thing to note is that at the time of writing, MySQL's transactional system is not `data definition language` aware, meaning that changes to table structures occurring during some backup schemes can result in corrupted backup copies. If schema changes will be occurring during your backup window, it is a good idea to ensure that appropriate locking mechanisms are used to prevent these changes during critical steps of the backup process.

However, on busy installations which cannot be stopped, the use of locks in many backup utilities may cause issues if your programs expect to write data to the database during the backup window.

In such cases it might make sense to review the given backup tools for locking related options or to use other mechanisms such as replicas or alternate backup tools to prevent interaction of the database.

2.6.6 Replication and snapshots for backup

Larger databases consisting of many Terabytes of data may take many hours or even days to backup and restore, and so downtime resulting from system failure can create major impacts to ongoing work.

While not backup tools per-se, use of MySQL master-slave replication and disk snapshots can be useful to assist in reducing the downtime resulting from a full database outage.

Replicas can be configured so that one copy of the data is immediately online in the event of server crash. When a server fails in this case, users and programs simply restart and point to the new server before resuming work.

Replicas can also reduce the system load generated by regular backup procedures, since they can be backed up instead of the main server. Additionally they can allow more flexibility in a given backup scheme, such as allowing for disk snapshots on a busy system that would not otherwise be able to be stopped. A replica copy can be stopped temporarily and then resumed while a disk snapshot or other backup operation occurs.

CHAPTER

THREE

CLIENT SETUP

3.1 Install and Connect

DataJoint is implemented for Python 3.4+. You may install it from PyPI:

```
pip3 install datajoint
```

or upgrade

```
pip3 install --upgrade datajoint
```

Next configure the connection through DataJoint's `config` object:

```
In [1]: import datajoint as dj
DataJoint 0.4.9 (February 1, 2017)
No configuration found. Use `dj.config` to configure and save the configuration.
```

You may now set the database credentials:

```
In [2]: dj.config['database.host'] = "alicelab.datajoint.io"
In [3]: dj.config['database.user'] = "alice"
In [4]: dj.config['database.password'] = "haha not my real password"
```

Skip setting the password to make DataJoint prompt to enter the password every time.

You may save the configuration in the local work directory with `dj.config.save_local()` or for all your projects in `dj.config.save_global()`. Configuration changes should be made through the `dj.config` interface; the config file should not be modified directly by the user.

You may leave the user or the password as `None`, in which case you will be prompted to enter them manually for every session. Setting the password as an empty string allows access without a password.

Note that the system environment variables `DJ_HOST`, `DJ_USER`, and `DJ_PASS` will overwrite the settings in the config file. You can use them to set the connection credentials instead of config files.

To change the password, the `dj.set_password` function will walk you through the process:

```
>>> dj.set_password()
```

After that, update the password in the configuration and save it as described above:

```
dj.config['database.password'] = 'my#cool!new*psswrd'
dj.config.save_local()   # or dj.config.save_global()
```

3.2 Other Configuration Settings

If you are not using DataJoint on your own, or are setting up a DataJoint system for other users, some additional configuraiton options may be required to support *TLS* or *external storage* .

3.2.1 TLS Configuration

Starting with v0.12 (Python) and v3.3.1 (MATLAB), DataJoint will by default use TLS if it is available. TLS can be forced on or off with the boolean `use_tls` in MATLAB, or `dj.config['database.use_tls']` in Python.

3.3 DataJoint Python Windows Install Guide

This document outlines the steps necessary to install DataJoint on Windows for use in connecting to a remote server hosting a DataJoint database. Some limited discussion of installing MySQL is discussed in *MySQL for Windows*, but is not covered in-depth since this is an uncommon usage scenario and not strictly required to connect to DataJoint pipelines.

3.3.1 Quick steps

Quick install steps for advanced users are as follows:

- Install latest Python 3.x and ensure it is in `PATH` (3.6.3 current at time of writing)
- `pip install datajoint`

For ERD drawing support:

- Install Graphviz for Windows and ensure it is in `PATH` (64 bit builds currently tested; URL below.)
- `pip install pydotplus matplotlib`

Detailed instructions follow.

3.3.2 Step 1: install Python

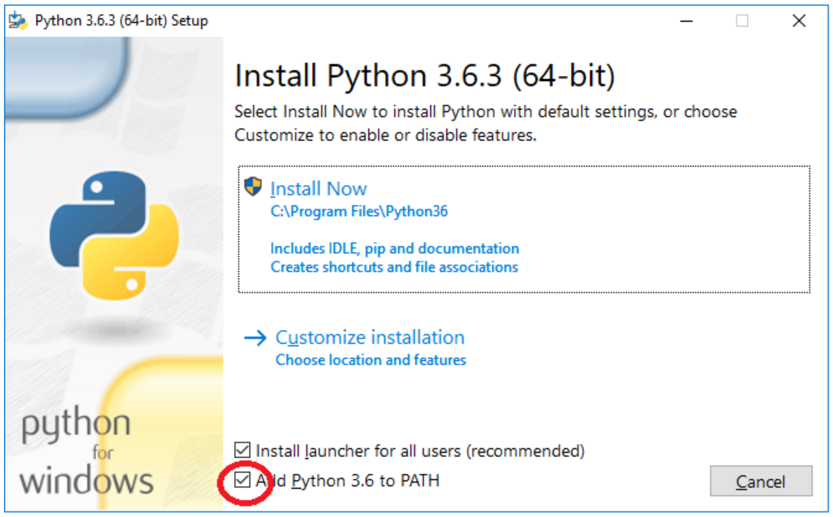
Python for Windows is available from:

https://www.python.org/downloads/windows

The latest 64 bit 3.x version, currently 3.6.3, is available from the Python site.

From here run the installer to install Python.

For a single-user machine, the regular installation process is sufficient - be sure to select the `Add Python to PATH` option:


Python 3.6.3 (64-bit) Setup

Install Python 3.6.3 (64-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

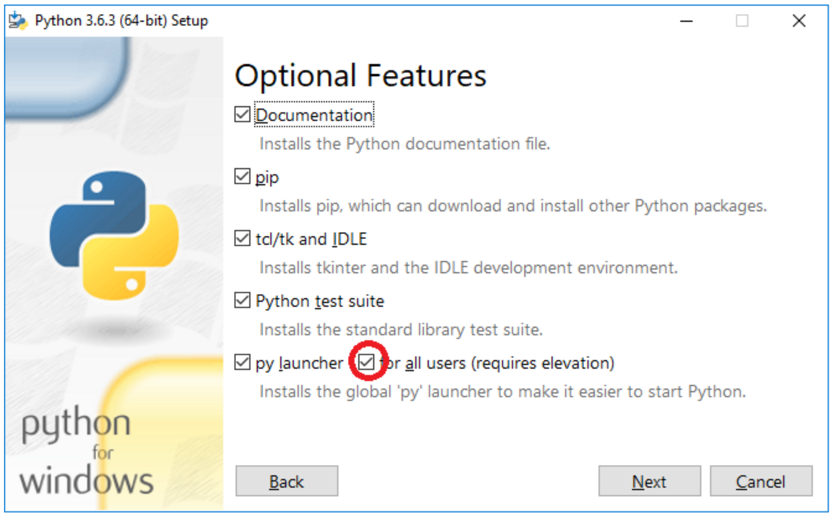
Install Now
C:\Program Files\Python36

Includes IDLE, pip and documentation
Creates shortcuts and file associations

Customize installation
Choose location and features

☑ Install launcher for all users (recommended)
☑ Add Python 3.6 to PATH

For a shared machine, run the installer as administrator (right-click, run as administrator) and select the advanced installation. Be sure to select options as follows:


Python 3.6.3 (64-bit) Setup

Optional Features

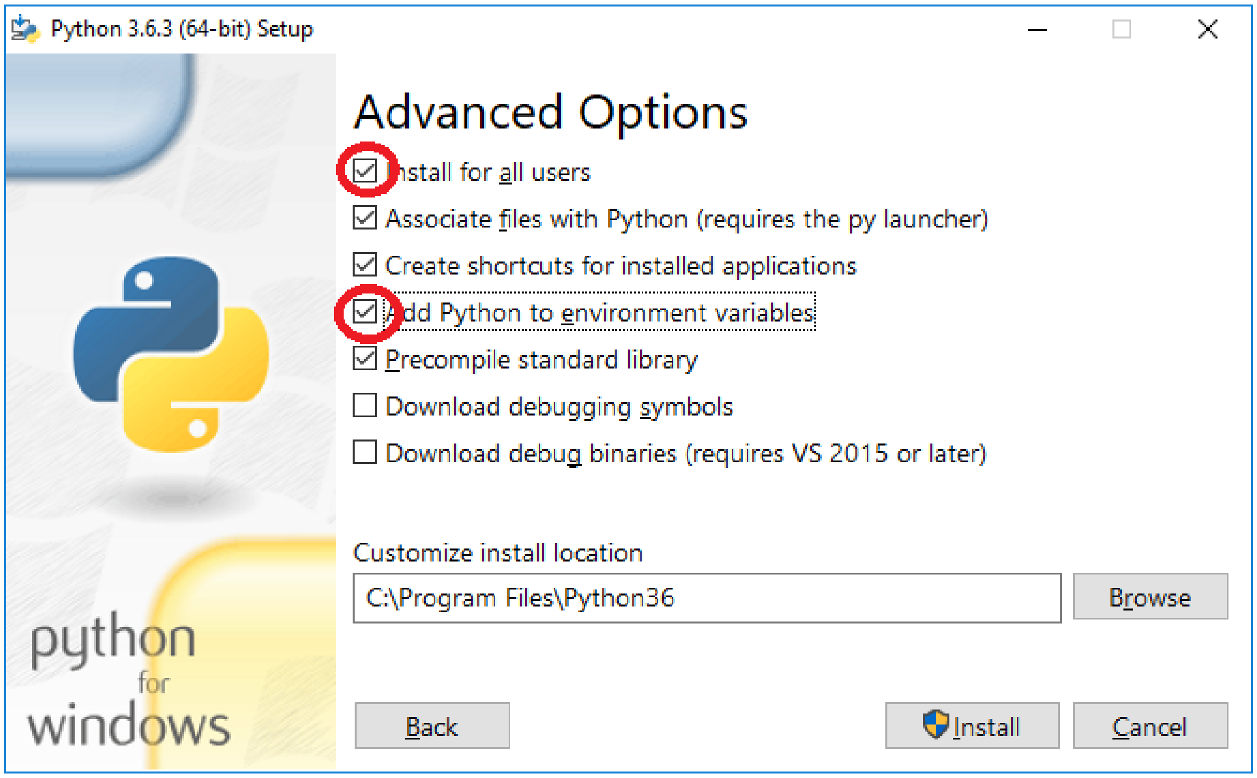
☑ Documentation
Installs the Python documentation file.

☑ pip
Installs pip, which can download and install other Python packages.

☑ tcl/tk and IDLE
Installs tkinter and the IDLE development environment.

☑ Python test suite
Installs the standard library test suite.

☑ py launcher ☑ for all users (requires elevation)
Installs the global 'py' launcher to make it easier to start Python.


Python 3.6.3 (64-bit) Setup

Advanced Options

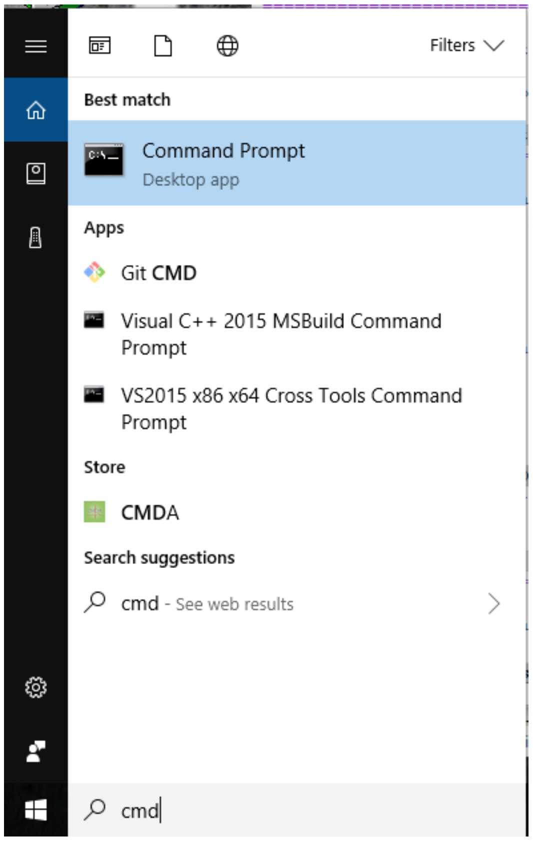
- ☑ Install for all users
- ☑ Associate files with Python (requires the py launcher)
- ☑ Create shortcuts for installed applications
- ☑ Add Python to environment variables
- ☑ Precompile standard library
- ☐ Download debugging symbols
- ☐ Download debug binaries (requires VS 2015 or later)

Customize install location

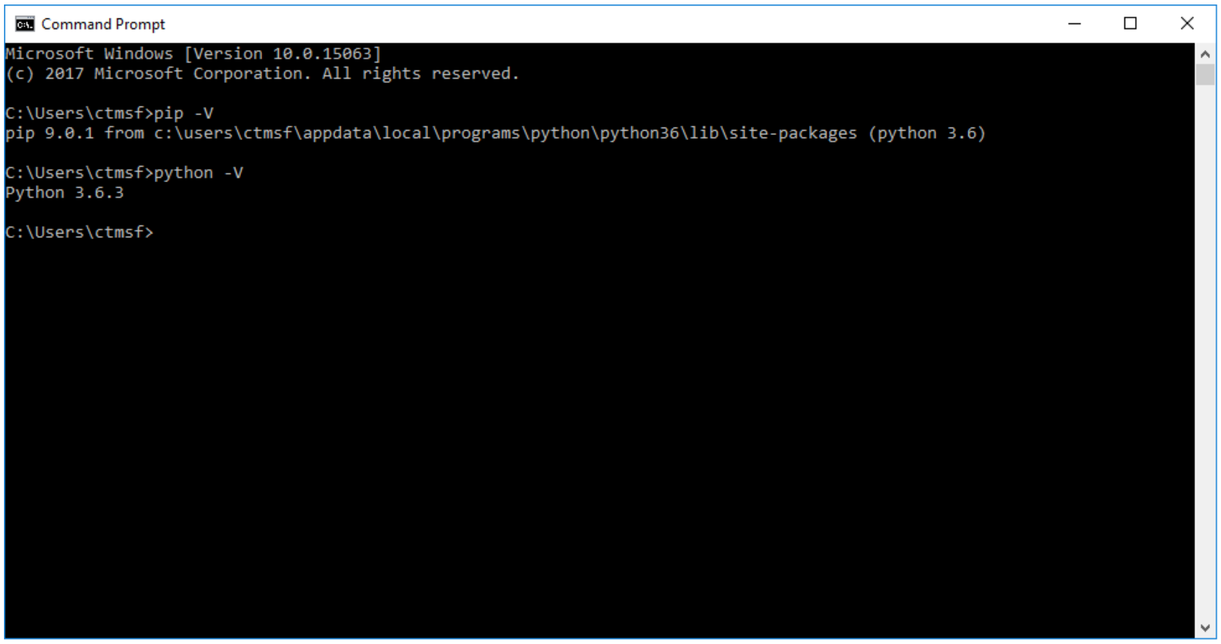
C:\Program Files\Python36

3.3.3 Step 2: verify installation

To verify the Python installation and make sure that your system is ready to install DataJoint, open a command window by entering `cmd` into the Windows search bar:



From here `python` and the Python package manager `pip` can be verified by running `python -V` and `pip -V`, respectively:



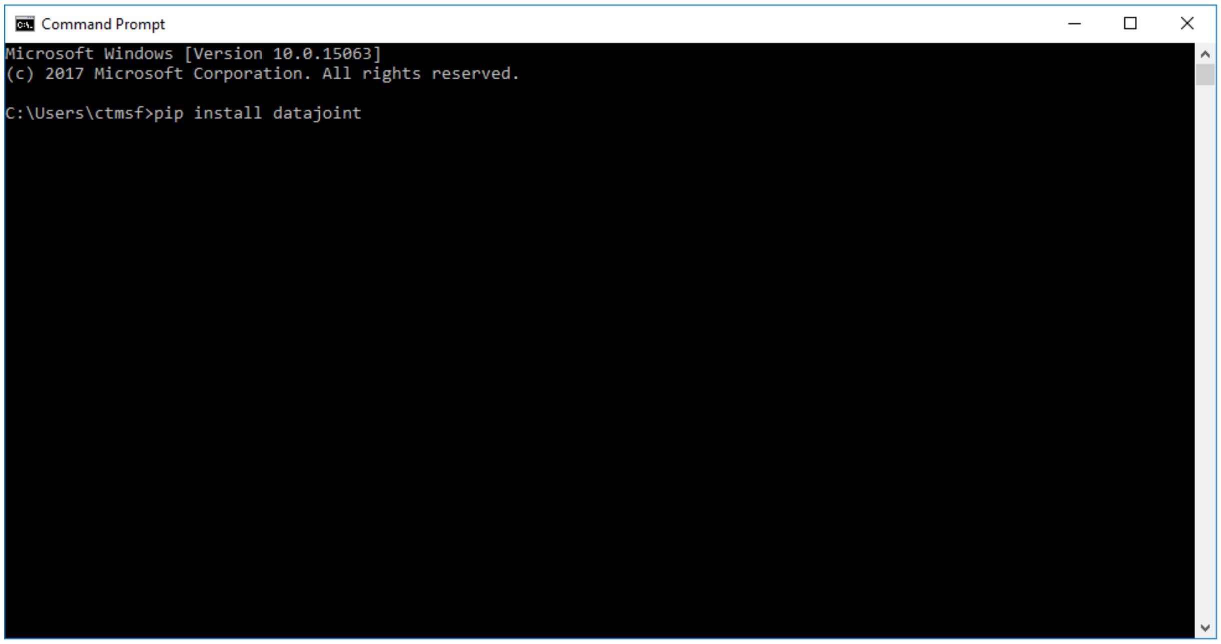
```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\ctmsf>pip -V
pip 9.0.1 from c:\users\ctmsf\appdata\local\programs\python\python36\lib\site-packages (python 3.6)

C:\Users\ctmsf>python -V
Python 3.6.3

C:\Users\ctmsf>
```

If you receive the error message that either `pip` or `python` is not a recognized command, please uninstall Python and ensure that the option to add Python to the `PATH` variable was properly configured.

3.3.4 Step 3: install DataJoint

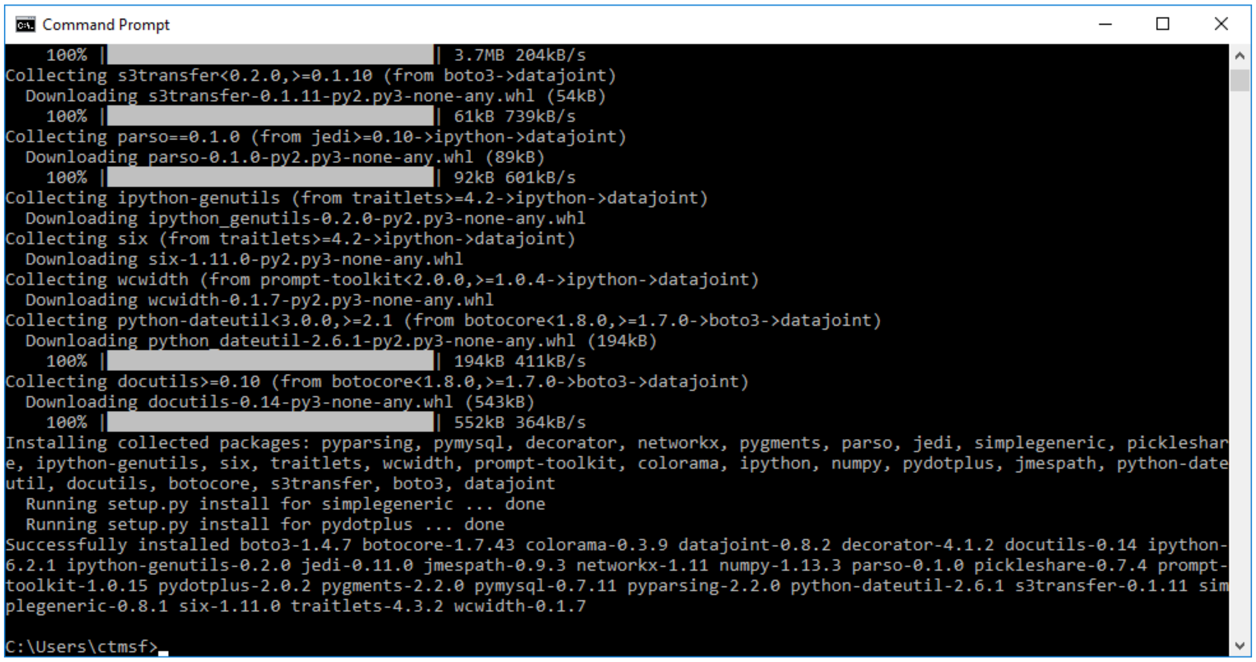
DataJoint (and other Python modules) can be easily installed using the `pip` Python package manager which is installed as a part of Python and was verified in the previous step.

To install DataJoint simply run `pip install datajoint`:



```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\ctmsf>pip install datajoint
```

This will proceed to install DataJoint, along with several other required packages from the PIP repository. When finished, a summary of the activity should be presented:



```
   100% |████████████████████████████████| 3.7MB 204kB/s
Collecting s3transfer<0.2.0,>=0.1.10 (from boto3->datajoint)
  Downloading s3transfer-0.1.11-py2.py3-none-any.whl (54kB)
   100% |████████████████████████████████| 61kB 739kB/s
Collecting parso==0.1.0 (from jedi>=0.10->ipython->datajoint)
  Downloading parso-0.1.0-py2.py3-none-any.whl (89kB)
   100% |████████████████████████████████| 92kB 601kB/s
Collecting ipython-genutils (from traitlets>=4.2->ipython->datajoint)
  Downloading ipython_genutils-0.2.0-py2.py3-none-any.whl
Collecting six (from traitlets>=4.2->ipython->datajoint)
  Downloading six-1.11.0-py2.py3-none-any.whl
Collecting wcwidth (from prompt-toolkit<2.0.0,>=1.0.4->ipython->datajoint)
  Downloading wcwidth-0.1.7-py2.py3-none-any.whl
Collecting python-dateutil<3.0.0,>=2.1 (from botocore<1.8.0,>=1.7.0->boto3->datajoint)
  Downloading python_dateutil-2.6.1-py2.py3-none-any.whl (194kB)
   100% |████████████████████████████████| 194kB 411kB/s
Collecting docutils>=0.10 (from botocore<1.8.0,>=1.7.0->boto3->datajoint)
  Downloading docutils-0.14-py3-none-any.whl (543kB)
   100% |████████████████████████████████| 552kB 364kB/s
Installing collected packages: pyparsing, pymysql, decorator, networkx, pygments, parso, jedi, simplegeneric, picklesha
e, ipython-genutils, six, traitlets, wcwidth, prompt-toolkit, colorama, ipython, numpy, pydotplus, jmespath, python-date
util, docutils, botocore, s3transfer, boto3, datajoint
  Running setup.py install for simplegeneric ... done
  Running setup.py install for pydotplus ... done
Successfully installed boto3-1.4.7 botocore-1.7.43 colorama-0.3.9 datajoint-0.8.2 decorator-4.1.2 docutils-0.14 ipython-
6.2.1 ipython-genutils-0.2.0 jedi-0.11.0 jmespath-0.9.3 networkx-1.11 numpy-1.13.3 parso-0.1.0 pickleshare-0.7.4 prompt-
toolkit-1.0.15 pydotplus-2.0.2 pygments-2.2.0 pymysql-0.7.11 pyparsing-2.2.0 python-dateutil-2.6.1 s3transfer-0.1.11 sim
plegeneric-0.8.1 six-1.11.0 traitlets-4.3.2 wcwidth-0.1.7

C:\Users\ctmsf>
```

Note: You can find out more about the packages installed here and many other freely available open source packages via pypi, the Python package index site.

3.3.5 (Optional) step 4: install packages for ERD support

To draw ERD diagrams of your DataJoint schema, the following additional steps should be followed.

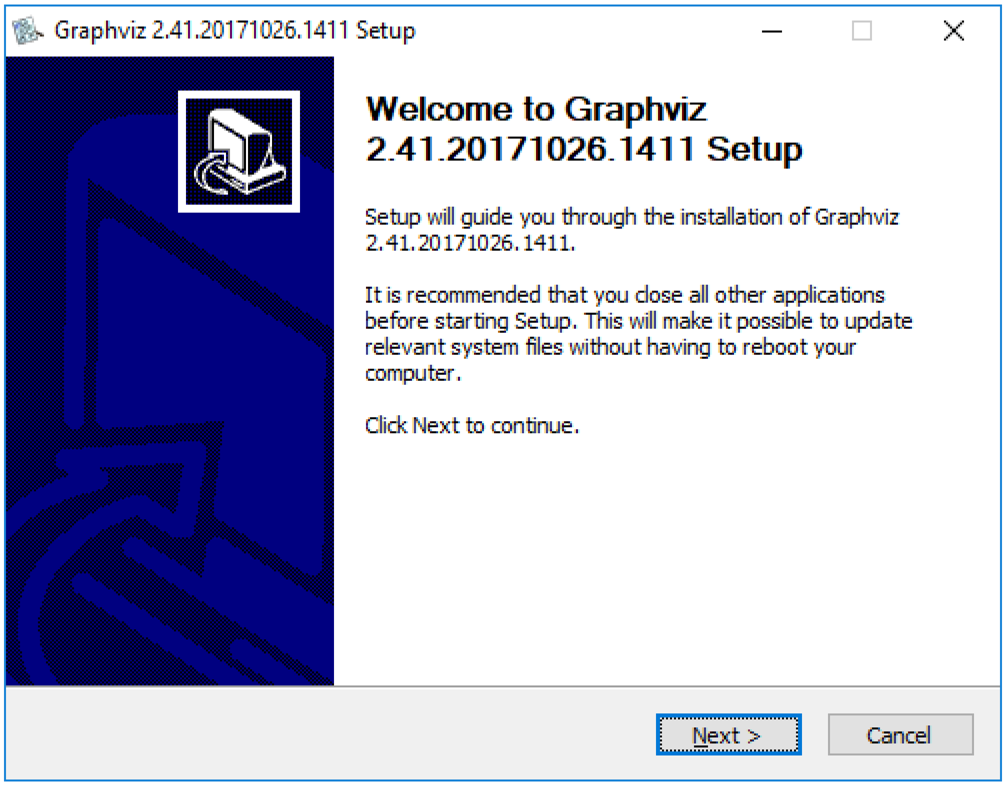
Install Graphviz

DataJoint currently utilizes Graphviz to generate the ERD visualizations. Although a Windows version of Graphviz is available from the main site, it is an older and out of date 32-bit version. The recommended pre-release builds of the 64 bit version are available here:

https://ci.appveyor.com/project/ellson/graphviz-pl238

More specifically, the build artifacts from the `Win64; Configuration:  Release` are recommended, available here.

This is a regular Windows installer executable, and will present a dialog when starting:



Graphviz 2.41.20171026.1411 Setup

Welcome to Graphviz 2.41.20171026.1411 Setup

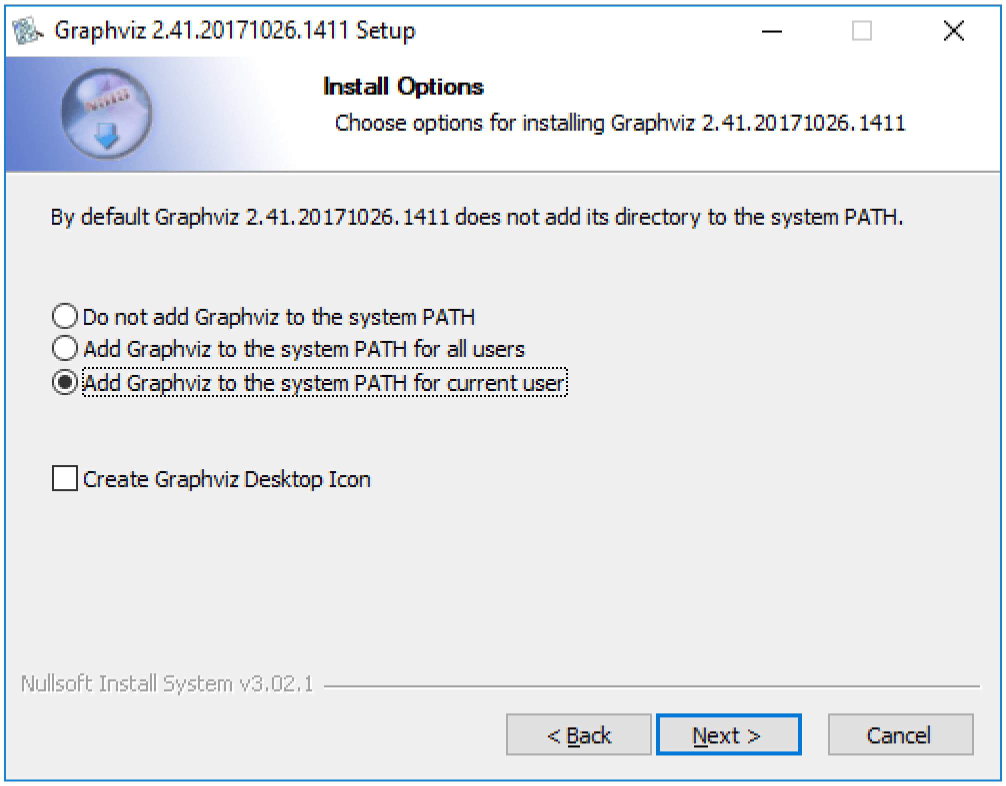
Setup will guide you through the installation of Graphviz 2.41.20171026.1411.

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.

It is important that an option to place Graphviz in the `PATH` be selected.

For a personal installation:



Graphviz 2.41.20171026.1411 Setup

Install Options

Choose options for installing Graphviz 2.41.20171026.1411

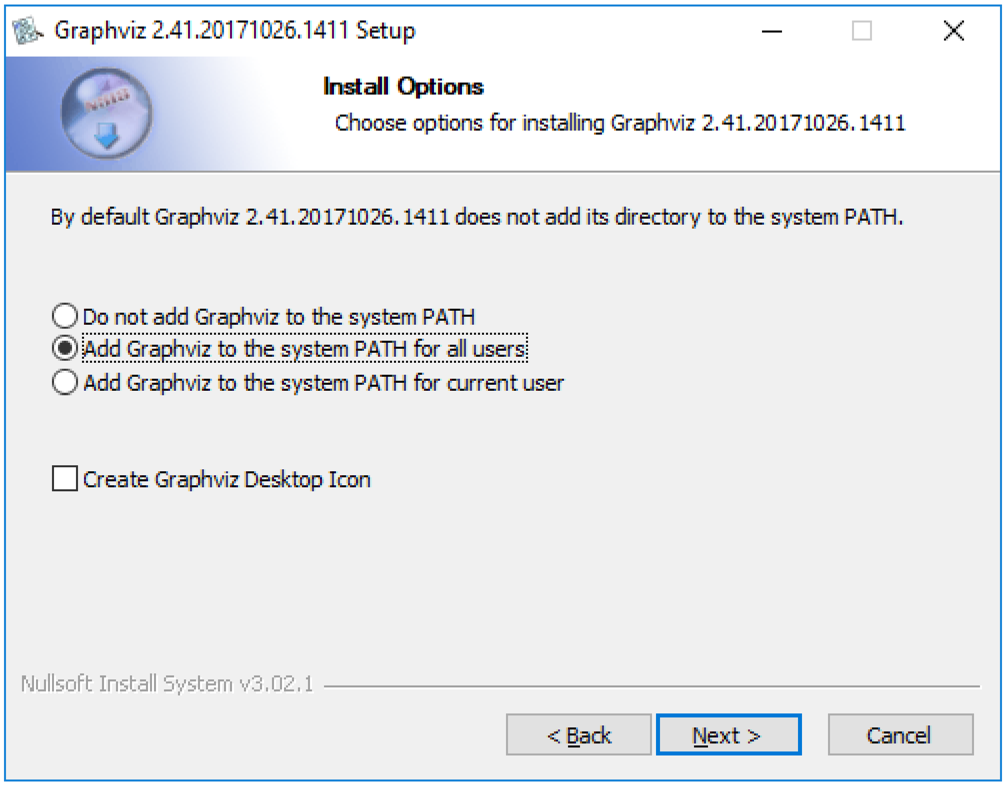
By default Graphviz 2.41.20171026.1411 does not add its directory to the system PATH.

- ○ Do not add Graphviz to the system PATH
- ○ Add Graphviz to the system PATH for all users
- ● Add Graphviz to the system PATH for current user

☐ Create Graphviz Desktop Icon

Nullsoft Install System v3.02.1

To install system wide:


Graphviz 2.41.20171026.1411 Setup

Install Options

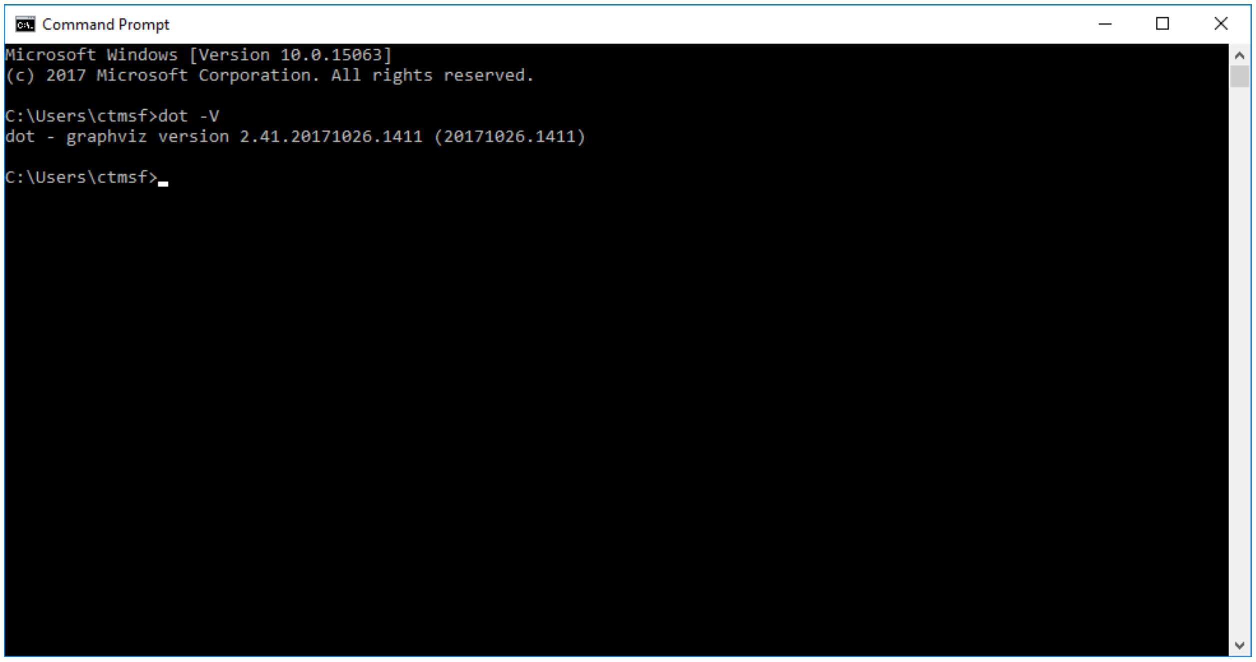
Choose options for installing Graphviz 2.41.20171026.1411

By default Graphviz 2.41.20171026.1411 does not add its directory to the system PATH.

- ○ Do not add Graphviz to the system PATH
- ● Add Graphviz to the system PATH for all users
- ○ Add Graphviz to the system PATH for current user

☐ Create Graphviz Desktop Icon

Once installed, Graphviz can be verified from a fresh command window as follows:



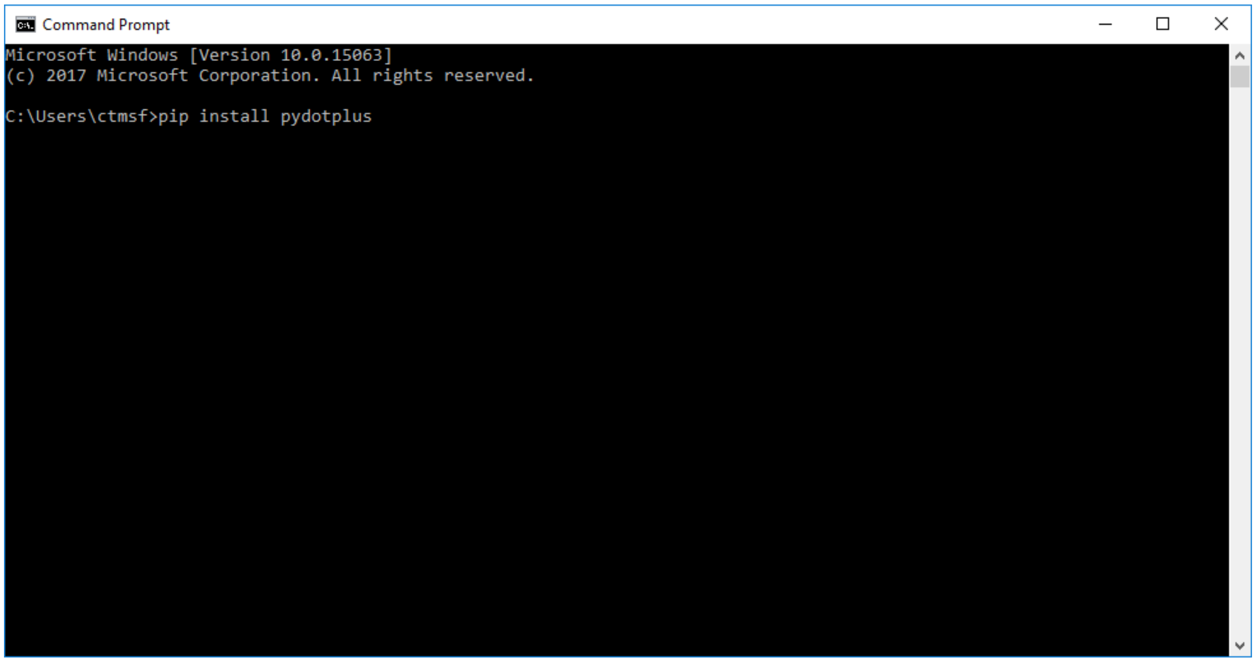
```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\ctmsf>dot -V
dot - graphviz version 2.41.20171026.1411 (20171026.1411)

C:\Users\ctmsf>_
```

If you receive the error message that the `dot` program is not a recognized command, please uninstall Graphviz and ensure that the option to add Python to the PATH variable was properly configured.

Important: in some cases, running the `dot -c` command in a command prompt is required to properly initialize the Graphviz installation.

Install PyDotPlus

The PyDotPlus library links the Graphviz installation to DataJoint and is easily installed via `pip` :

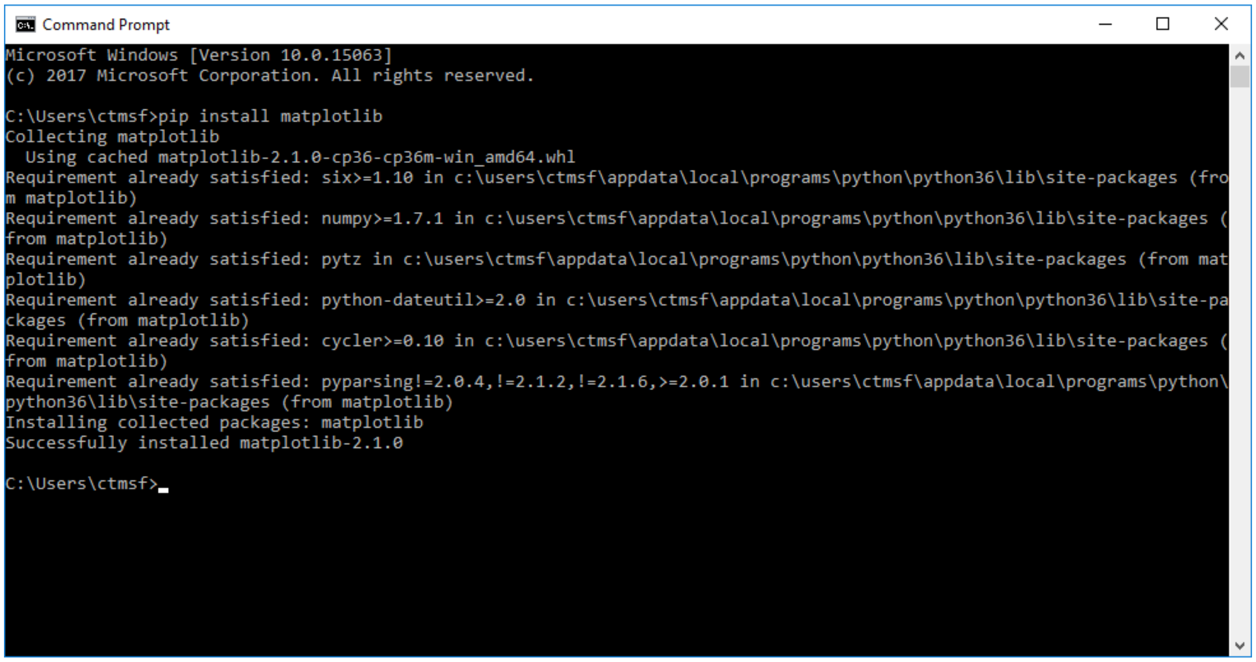


Command Prompt

```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\ctmsf>pip install pydotplus
```

Install Matplotlib

The Matplotlib library provides useful plotting utilities which are also used by DataJoint's ERD drawing facility. The package is easily installed via `pip` :



Command Prompt

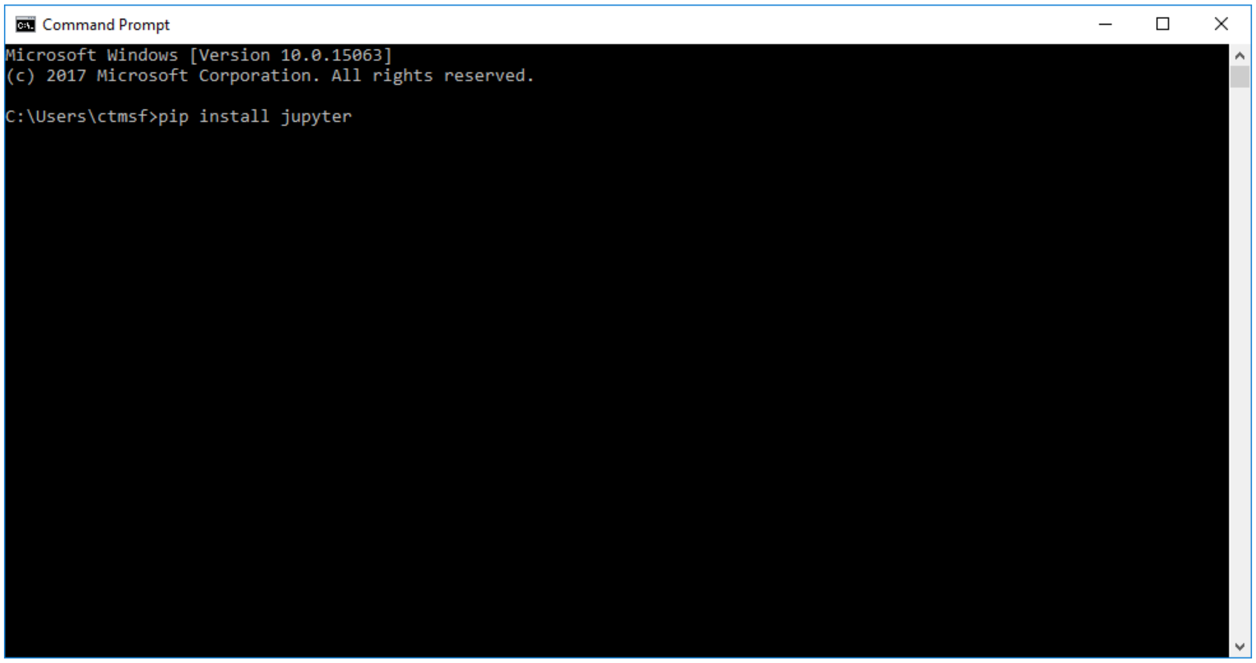
```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\ctmsf>pip install matplotlib
Collecting matplotlib
  Using cached matplotlib-2.1.0-cp36-cp36m-win_amd64.whl
Requirement already satisfied: six>=1.10 in c:\users\ctmsf\appdata\local\programs\python\python36\lib\site-packages (fro
m matplotlib)
Requirement already satisfied: numpy>=1.7.1 in c:\users\ctmsf\appdata\local\programs\python\python36\lib\site-packages (
from matplotlib)
Requirement already satisfied: pytz in c:\users\ctmsf\appdata\local\programs\python\python36\lib\site-packages (from mat
plotlib)
Requirement already satisfied: python-dateutil>=2.0 in c:\users\ctmsf\appdata\local\programs\python\python36\lib\site-pa
ckages (from matplotlib)
Requirement already satisfied: cycler>=0.10 in c:\users\ctmsf\appdata\local\programs\python\python36\lib\site-packages (
from matplotlib)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in c:\users\ctmsf\appdata\local\programs\python\
python36\lib\site-packages (from matplotlib)
Installing collected packages: matplotlib
Successfully installed matplotlib-2.1.0

C:\Users\ctmsf>_
```

3.3.6 (Optional) step 5: install Jupyter Notebook

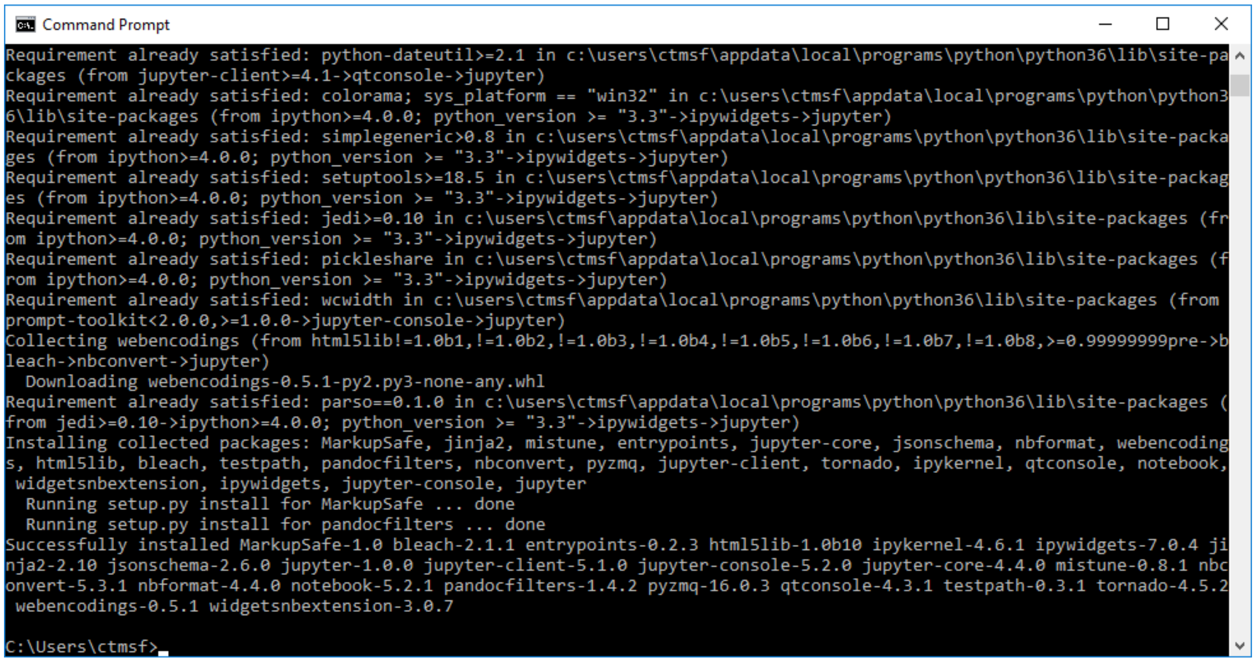
As described on the jupyter.org website:

> 'The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text.'

Although not a part of DataJoint, Jupyter Notebook can be a very useful tool for building and interacting with DataJoint pipelines. It is easily installed from `pip` as well:

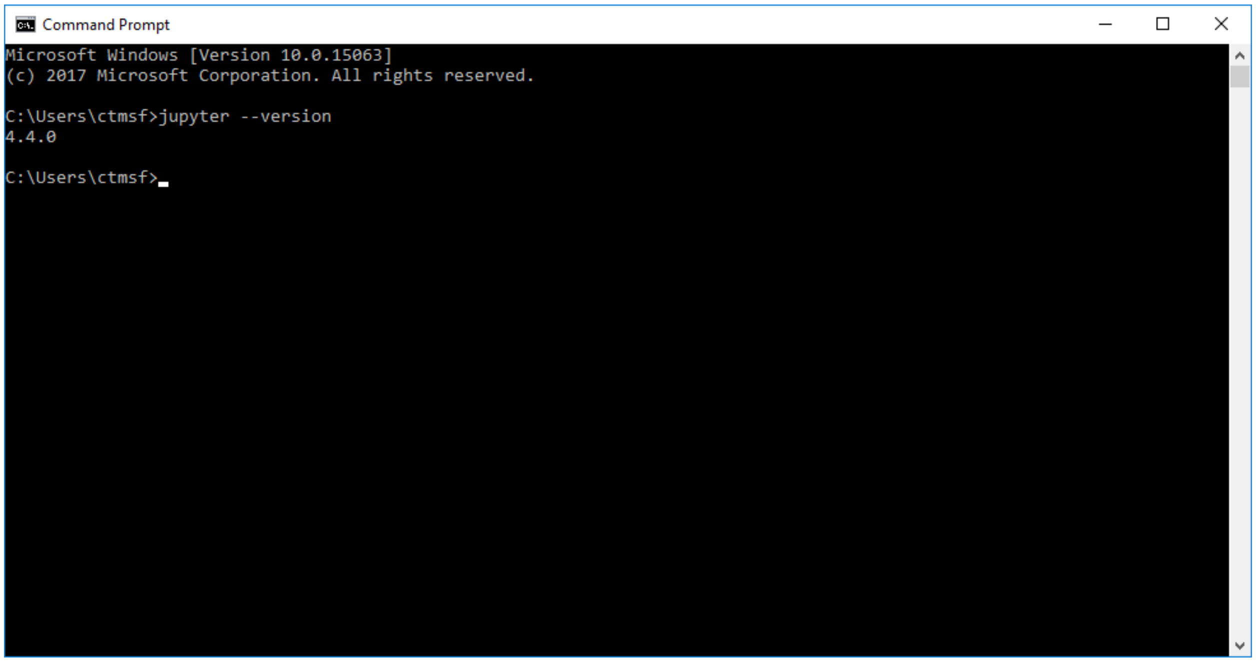


```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\ctmsf>pip install jupyter
```



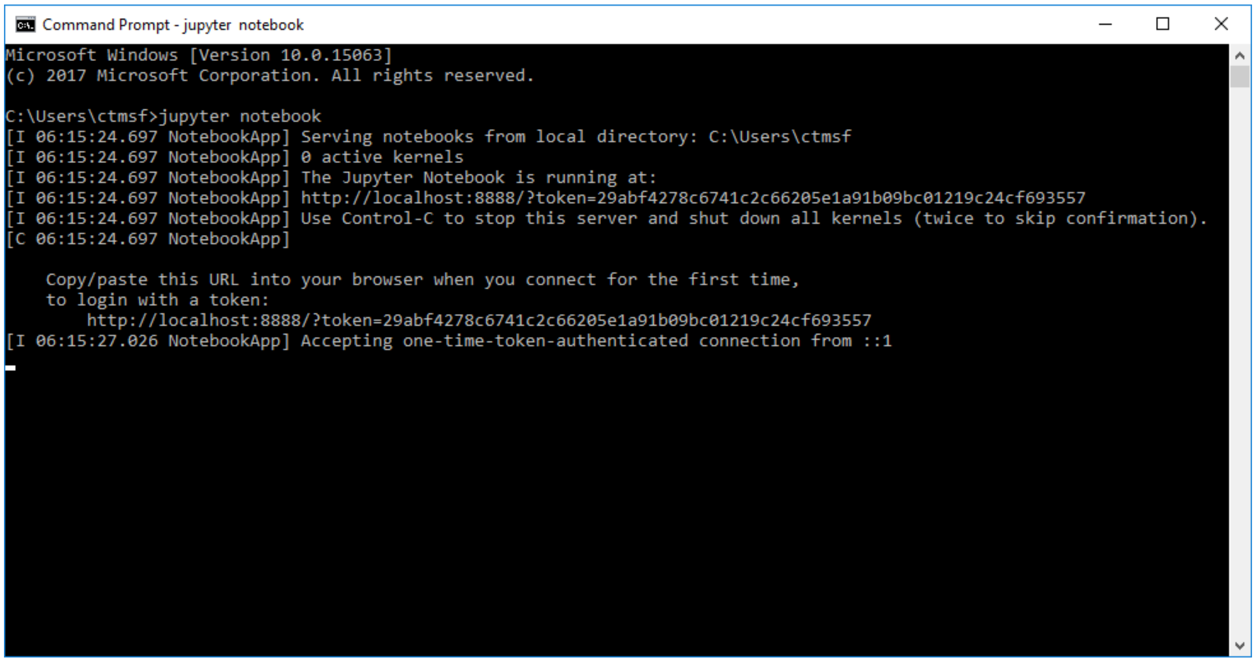
```
Requirement already satisfied: python-dateutil>=2.1 in c:\users\ctmsf\appdata\local\programs\python\python36\lib\site-pa
ckages (from jupyter-client>=4.1->qtconsole->jupyter)
Requirement already satisfied: colorama; sys_platform == "win32" in c:\users\ctmsf\appdata\local\programs\python\python3
6\lib\site-packages (from ipython>=4.0.0; python_version >= "3.3"->ipywidgets->jupyter)
Requirement already satisfied: simplegeneric>0.8 in c:\users\ctmsf\appdata\local\programs\python\python36\lib\site-packa
ges (from ipython>=4.0.0; python_version >= "3.3"->ipywidgets->jupyter)
Requirement already satisfied: setuptools>=18.5 in c:\users\ctmsf\appdata\local\programs\python\python36\lib\site-packag
es (from ipython>=4.0.0; python_version >= "3.3"->ipywidgets->jupyter)
Requirement already satisfied: jedi>=0.10 in c:\users\ctmsf\appdata\local\programs\python\python36\lib\site-packages (fr
om ipython>=4.0.0; python_version >= "3.3"->ipywidgets->jupyter)
Requirement already satisfied: pickleshare in c:\users\ctmsf\appdata\local\programs\python\python36\lib\site-packages (f
rom ipython>=4.0.0; python_version >= "3.3"->ipywidgets->jupyter)
Requirement already satisfied: wcwidth in c:\users\ctmsf\appdata\local\programs\python\python36\lib\site-packages (from
prompt-toolkit<2.0.0,>=1.0.0->jupyter-console->jupyter)
Collecting webencodings (from html5lib!=1.0b1,!=1.0b2,!=1.0b3,!=1.0b4,!=1.0b5,!=1.0b6,!=1.0b7,!=1.0b8,>=0.99999999pre->b
leach->nbconvert->jupyter)
  Downloading webencodings-0.5.1-py2.py3-none-any.whl
Requirement already satisfied: parso==0.1.0 in c:\users\ctmsf\appdata\local\programs\python\python36\lib\site-packages (
from jedi>=0.10->ipython>=4.0.0; python_version >= "3.3"->ipywidgets->jupyter)
Installing collected packages: MarkupSafe, jinja2, mistune, entrypoints, jupyter-core, jsonschema, nbformat, webencoding
s, html5lib, bleach, testpath, pandocfilters, nbconvert, pyzmq, jupyter-client, tornado, ipykernel, qtconsole, notebook,
 widgetsnbextension, ipywidgets, jupyter-console, jupyter
  Running setup.py install for MarkupSafe ... done
  Running setup.py install for pandocfilters ... done
Successfully installed MarkupSafe-1.0 bleach-2.1.1 entrypoints-0.2.3 html5lib-1.0b10 ipykernel-4.6.1 ipywidgets-7.0.4 ji
nja2-2.10 jsonschema-2.6.0 jupyter-1.0.0 jupyter-client-5.1.0 jupyter-console-5.2.0 jupyter-core-4.4.0 mistune-0.8.1 nbc
onvert-5.3.1 nbformat-4.4.0 notebook-5.2.1 pandocfilters-1.4.2 pyzmq-16.0.3 qtconsole-4.3.1 testpath-0.3.1 tornado-4.5.2
 webencodings-0.5.1 widgetsnbextension-3.0.7

C:\Users\ctmsf>_
```

Once installed, Jupyter Notebook can be started via the `jupyter notebook` command, which should now be on your path:

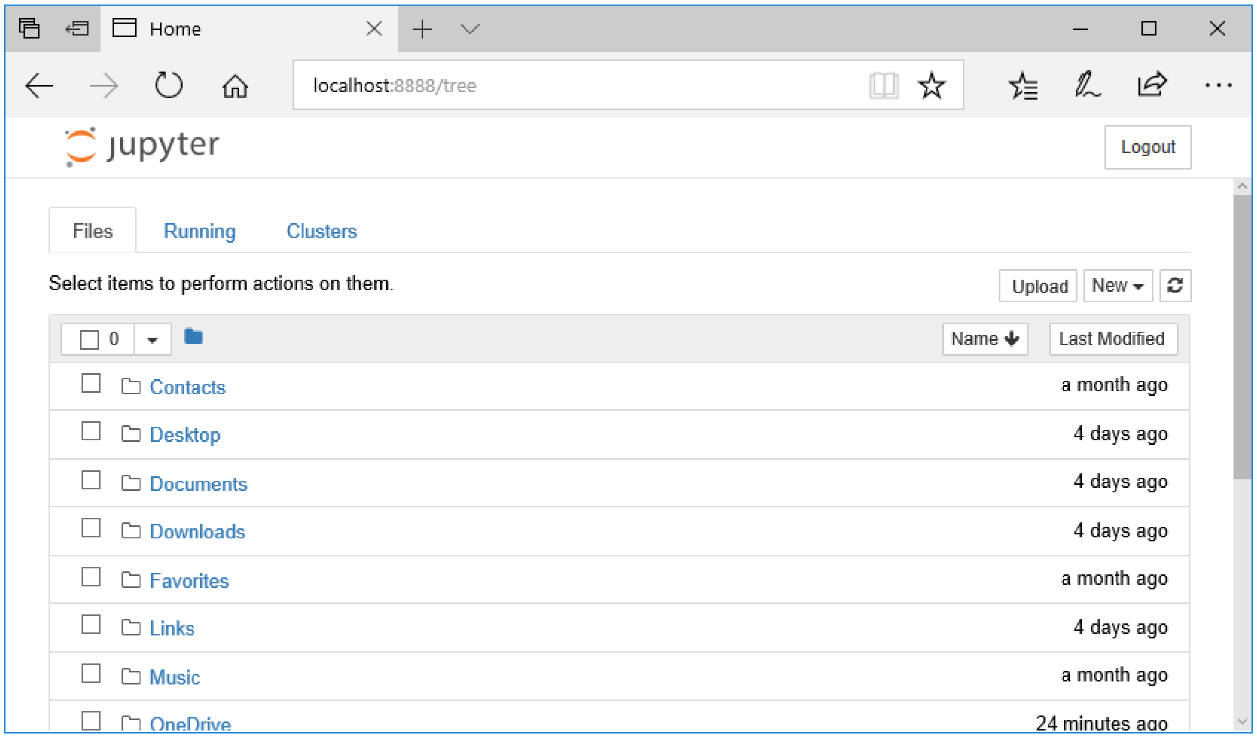


```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\ctmsf>jupyter --version
4.4.0

C:\Users\ctmsf>
```

By default Jupyter Notebook will start a local private webserver session from the directory where it was started and start a web browser session connected to the session.



```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\ctmsf>jupyter notebook
[I 06:15:24.697 NotebookApp] Serving notebooks from local directory: C:\Users\ctmsf
[I 06:15:24.697 NotebookApp] 0 active kernels
[I 06:15:24.697 NotebookApp] The Jupyter Notebook is running at:
[I 06:15:24.697 NotebookApp] http://localhost:8888/?token=29abf4278c6741c2c66205e1a91b09bc01219c24cf693557
[I 06:15:24.697 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 06:15:24.697 NotebookApp]

    Copy/paste this URL into your browser when you connect for the first time,
    to login with a token:
        http://localhost:8888/?token=29abf4278c6741c2c66205e1a91b09bc01219c24cf693557
[I 06:15:27.026 NotebookApp] Accepting one-time-token-authenticated connection from ::1
```



Select items to perform actions on them.

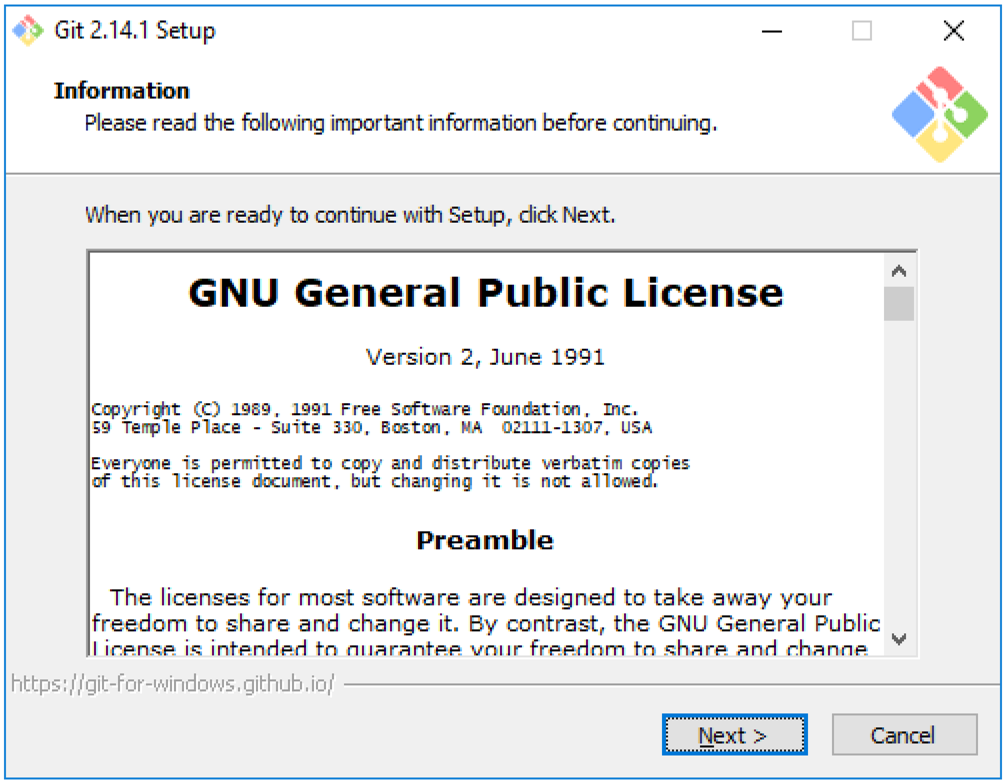
Name	Last Modified
Contacts	a month ago
Desktop	4 days ago
Documents	4 days ago
Downloads	4 days ago
Favorites	a month ago
Links	4 days ago
Music	a month ago
OneDrive	24 minutes ago

You now should be able to use the notebook viewer to navigate the filesystem and to create new project folders and interactive Jupyter/Python/DataJoint notebooks.

3.3.7 Git for Windows

The Git version control system is not a part of DataJoint but is recommended for interacting with the broader Python/Git/GitHub sharing ecosystem.

The Git for Windows installer is available from https://git-scm.com/download/win.



Git 2.14.1 Setup

Information

Please read the following important information before continuing.

When you are ready to continue with Setup, click Next.

GNU General Public License

Version 2, June 1991

```
Copyright (C) 1989, 1991 Free Software Foundation, Inc.
59 Temple Place - Suite 330, Boston, MA  02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.
```

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change

https://git-for-windows.github.io/

The default settings should be sufficient and correct in most cases.

3.3.8 MySQL for Windows

For hosting pipelines locally, the MySQL server package is required.

MySQL for windows can be installed via the installers available from the MySQL website. Please note that although DataJoint should be fully compatible with a Windows MySQL server installation, this mode of operation is not tested by the DataJoint team.

CHAPTER

FOUR

CONCEPTS

4.1 Data Model

4.1.1 What is a data model?

A **data model** refers to a conceptual framework for thinking about data and about operations on data. A data model defines the mental toolbox of the data scientist; it has less to do with the architecture of the data systems, although architectures are often intertwined with data models.

Among the most familiar data models are those based on files and folders: data of any kind are lumped together into binary strings called **files**, files are collected into folders, and folders can be nested within other folders to create a folder hierarchy.

Another family of data models are various **tabular models**. For example, items in CSV files are listed in rows, and the attributes of each item are stored in columns. Various **spreadsheet** models allow forming dependencies between cells and groups of cells, including complex calculations.

The **object data model** is common in programming, where data are represented as objects in memory with properties and methods for transformations of such data.

4.1.2 Relational data model

The **relational model** is a way of thinking about data as sets and operations on sets. Formalized almost a half-century ago (Codd, 1969), the relational data model provides the most rigorous approach to structured data storage and the most precise approach to data querying. The model is defined by the principles of data representation, domain constraints, uniqueness constraints, referential constraints, and declarative queries as summarized below.

Core principles of the relational data model

Data representation. Data are represented and manipulated in the form of relations. A relation is a set (i.e. an unordered collection) of entities of values for each of the respective named attributes of the relation. Base relations represent stored data while derived relations are formed from base relations through query expressions. A collection of base relations with their attributes, domain constraints, uniqueness constraints, and referential constraints is called a schema.

Domain constraints. Attribute values are drawn from corresponding attribute domains, i.e. predefined sets of values. Attribute domains may not include relations, which keeps the data model flat, i.e. free of nested structures.

Uniqueness constraints. Entities within relations are addressed by values of their attributes. To identify and relate data elements, uniqueness constraints are imposed on subsets of attributes. Such subsets are then referred to as keys. One key in a relation is designated as the primary key used for referencing its elements.

Referential constraints. Associations among data are established by means of referential constraints with the help of foreign keys. A referential constraint on relation A referencing relation B allows only those entities in A whose foreign key attributes match the key attributes of an entity in B.

Declarative queries. Data queries are formulated through declarative, as opposed to imperative, specifications of sought results. This means that query expressions convey the logic for the result rather than the procedure for obtaining it. Formal languages for query expressions include relational algebra, relational calculus, and SQL.

The relational model has many advantages over both hierarchical file systems and tabular models for maintaining data integrity and providing flexible access to interesting subsets of the data.

Popular implementations of the relational data model rely on the Structured Query Language (SQL). SQL comprises distinct sublanguages for schema definition, data manipulation, and data queries. SQL thoroughly dominates in the space of relational databases and is often conflated with the relational data model in casual discourse. Various terminologies are used to describe related concepts from the relational data model. Similar to spreadsheets, relations are often visualized as tables with *attributes* corresponding to *columns* and *entities* corresponding to *rows*. In particular, SQL uses the terms *table*, *column*, and *row*.

4.1.3 DataJoint is a refinement of the relational data model

DataJoint is a conceptual refinement of the relational data model offering a more expressive and rigorous framework for database programming (Yatsenko et al., 2018). The DataJoint model facilitates clear conceptual modeling, efficient schema design, and precise and flexible data queries. The model has emerged over a decade of continuous development of complex data pipelines for neuroscience experiments (Yatsenko et al., 2015). DataJoint has allowed researchers with no prior knowledge of databases to collaborate effectively on common data pipelines sustaining data integrity and supporting flexible access. DataJoint is currently implemented as client libraries in MATLAB and Python. These libraries work by transpiling DataJoint queries into SQL before passing them on to conventional relational database systems that serve as the backend, in combination with bulk storage systems for storing large contiguous data objects.

DataJoint comprises:

- a schema *definition* language
- a data *manipulation* language
- a data *query* language
- a *diagramming* notation for visualizing relationships between modeled entities

The key refinement of DataJoint over other relational data models and their implementations is DataJoint's support of *entity normalization*.

4.2 Terminology

DataJoint introduces a principled data model, which is described in detail in Yatsenko et al., 2018. This data model is a conceptual refinement of the Relational Data Model and also draws on the Entity-Relationship Model (ERM).

The Relational Data Model was inspired by the concepts of relations in Set Theory. When the formal relational data model was formulated, it introduced additional terminology (e.g. *relation*, *attribute*, *tuple*, *domain*). Practical programming languages such as SQL do not precisely follow the relational data model and introduce other terms to approximate relational concepts (e.g. *table*, *column*, *row*, *datatype*). Subsequent data models (e.g. ERM) refined the relational data model and introduced their own terminology to describe analogous concepts (e.g. *entity set*, *relationship set*, *attribute set*). As a result, similar concepts may be described using different sets of terminologies, depending on the context and the speaker's background.

For example, what is known as a **relation** in the formal relational model is called a **table** in SQL; the analogous concept in ERM and DataJoint is called an **entity set**.

The DataJoint documentation follows the terminology defined in Yatsenko et al, 2018, except *entity set* is replaced with the more colloquial *table* or *query result* in most cases.

The table below summarizes the terms used for similar concepts across the related data models.

Table 1: Data model terminology

Relational	ERM	SQL	DataJoint (formal)	This manual
relation	entity set	table	entity set	table
tuple	entity	row	entity	entity
domain	value set	datatype	datatype	datatype
attribute	attribute	column	attribute	attribute
attribute value	attribute value	field value	attribute value	attribute value
primary key	primary key	primary key	primary key	primary key
foreign key	foreign key	foreign key	foreign key	foreign key
schema	schema	schema or database	schema	schema
relational expression	data query	`SELECT` statement	query expression	query expression

4.2.1 DataJoint: databases, schemas, packages, and modules

A **database** is collection of tables on the database server. DataJoint users do not interact with it directly.

A **DataJoint schema** is

- a database on the database server containing tables with data *and*
- a collection of classes (in MATLAB or Python) associated with the database, one class for each table.

In MATLAB, the collection of classes is organized as a **package**, i.e. a file folder starting with a `+`.

In Python, the collection of classes is any set of classes decorated with the appropriate `schema` object. Very commonly classes for tables in one database are organized as a distinct Python module. Thus, typical DataJoint projects have one module per database. However, this organization is up to the user's discretion.

4.2.2 Base tables

Base tables are tables stored in the database, and are often referred to simply as *tables* in DataJoint. Base tables are distinguished from **derived tables**, which result from relational *operators*.

4.2.3 Relvars and relation values

Early versions of the DataJoint documentation referred to the relation objects as **relvars** https://en.wikipedia.org/wiki/Relvar. This term emphasizes the fact that relational variables and expressions do not contain actual data but are rather symbolic representations of data to be retrieved from the database. The specific value of a relvar would then be referred to as the **relation value**. The value of a relvar can change with changes in the state of the database.

The more recent iteration of the documentation has grown less pedantic and more often uses the term *table* instead.

4.2.4 Metadata

The vocabulary of DataJoint does not include this term.

In data science, the term **metadata** commonly means “data about the data” rather than the data themselves. For example, metadata could include data sizes, timestamps, data types, indexes, keywords.

In contrast, neuroscientists often use the term to refer to conditions and annotations about experiments. This distinction arose when such information was stored separately from experimental recordings, such as in physical notebooks. Such “metadata” are used to search and to classify the data and are in fact an integral part of the *actual* data.

In DataJoint, all data other than blobs can be used in searches and categorization. These fields may originate from manual annotations, preprocessing, or analyses just as easily as from recordings or behavioral performance. Since “metadata” in the neuroscience sense are not distinguished from any other data in a pipeline, DataJoint avoids the term entirely. Instead, DataJoint differentiates data into *data tiers*.

4.3 Entity Normalization

DataJoint uses a uniform way of representing any data. It does so in the form of **entity sets**, unordered collections of entities of the same type. The term **entity normalization** describes the commitment to represent all data as well-formed entity sets. Entity normalization is a conceptual refinement of the *relational data model* and is the central principle of the DataJoint model (Yatsenko et al., 2018). Entity normalization leads to clear and logical database designs and to easily comprehensible data queries.

Entity sets are a type of **relation** (from the *relational data model*) and are often visualized as **tables**. Hence the terms **relation**, **entity set**, and **table** can be used interchangeably when entity normalization is assumed.

4.3.1 Criteria of a well-formed entity set

1. All elements of an entity set belong to the same well-defined and readily identified **entity type** from the model world.
2. All attributes of an entity set are applicable directly to each of its elements, although some attribute values may be missing (set to null).
3. All elements of an entity set must be distinguishable form each other by the same primary key.
4. Primary key attribute values cannot be missing, i.e. set to null.
5. All elements of an entity set participate in the same types of relationships with other entity sets.

4.3.2 Entity normalization in schema design

Entity normalization applies to schema design in that the designer is responsible for the identification of the essential entity types in their model world and of the dependencies among the entity types.

The term entity normalization may also apply to a procedure for refactoring a schema design that does not meet the above criteria into one that does. In some cases, this may require breaking up some entity sets into multiple entity sets, which may cause some entities to be represented across multiple entity sets. In other cases, this may require converting attributes into their own entity sets. Technically speaking, entity normalization entails compliance with the Boyce-Codd normal form while lacking the representational power for the applicability of more complex normal forms (Kent, 1983). Adherence to entity normalization prevents redundancies in storage and data manipulation anomalies. The same criteria originally motivated the formulation of the classical relational normal forms.

4.3.3 Entity normalization in data queries

Entity normalization applies to data queries as well. DataJoint's *query operators* are designed to preserve the entity normalization of their inputs. For example, the outputs of operators *restriction*, *proj*, and *aggr* retain the same entity type as the (first) input. The *join* operator produces a new entity type comprising the pairing of the entity types of its inputs. *Universal sets* explicitly introduce virtual entity sets when necessary to accomplish a query.

4.3.4 Examples of poor normalization

Design choices lacking entity normalization may lead to data inconsistencies or anomalies. Below are several examples of poorly normalized designs and their normalized alternatives.

Indirect attributes

All attributes should apply to the entity itself. Avoid attributes that actually apply to one of the entity's other attributes. For example, consider the table `Author` with attributes `author_name`, `institution`, and `institution_address`. The attribute `institution_address` should really be held in a separate `Institution` table that `Author` depends on.

Repeated attributes

Avoid tables with repeated attributes of the same category. A better solution is to create a separate table that depends on the first (often a *part table*), with multiple individual entities rather than repeated attributes. For example, consider the table `Protocol` that includes the attributes ``equipment1`, ``equipment2``, and `equipment3`. A better design would be to create a `ProtocolEquipment` table that links each entity in `Protocol` with multiple entities in `Equipment` through *dependencies*.

Attributes that do not apply to all entities

All attributes should be relevant to every entity in a table. Attributes that apply only to a subset of entities in a table likely belong in a separate table containing only that subset of entities. For example, a table `Protocol` should include the attribute `stimulus` only if all experiment protocols include stimulation. If the not all entities in `Protocol` involve stimulation, then the `stimulus` attribute should be moved to a part table that has `Protocol` as its master. Only protocols using stimulation will have an entry in this part table.

Transient attributes

Attributes should be relevant to all entities in a table at all times. Attributes that do not apply to all entities should be moved to another dependent table containing only the appropriate entities. This principle also applies to attributes that have not yet become meaningful for some entities or that will not remain meaningful indefinitely. For example, consider the table `Mouse` with attributes `birth_date` and `death_date`, where `death_date` is set to `NULL` for living mice. Since the `death_date` attribute is not meaningful for mice that are still living, the proper design would include a separate table `DeceasedMouse` that depends on `Mouse`. `DeceasedMouse` would only contain entities for dead mice, which improves integrity and averts the need for *updates*.

4.4 Data Integrity

The term **data integrity** describes guarantees made by the data management process that prevent errors and corruption in data due to technical failures and human errors arising in the course of continuous use by multiple agents. DataJoint pipelines respect the following forms of data integrity: **entity integrity**, **referential integrity**, and **group integrity** as described in more detail below.

4.4.1 Entity integrity

In a proper relational design, each table represents a collection of discrete real-world entities of some kind. **Entity integrity** is the guarantee made by the data management process that entities from the real world are reliably and uniquely represented in the database system. Entity integrity states that the data management process must prevent duplicate representations or misidentification of entities. DataJoint enforces entity integrity through the use of *primary keys*.

Entity integrity breaks down when a process allows data pertaining to the same real-world entity to be entered into the database system multiple times. For example, a school database system may use unique ID numbers to distinguish students. Suppose the system automatically generates an ID number each time a student record is entered into the database without checking whether a record already exists for that student. Such a system violates entity integrity, because the same student may be assigned multiple ID numbers. The ID numbers succeed in uniquely identifying each student record but fail to do so for the actual students.

Note that a database cannot guarantee or enforce entity integrity by itself. Entity integrity is a property of the entire data management process as a whole, including institutional practices and user actions in addition to database configurations.

4.4.2 Referential integrity

Referential integrity is the guarantee made by the data management process that related data across the database remain present, correctly associated, and mutually consistent. Guaranteeing referential integrity means enforcing the constraint that no entity can exist in the database without all the other entities on which it depends. Referential integrity cannot exist without entity integrity: references to entity cannot be validated if the identity of the entity itself is not guaranteed.

Referential integrity fails when a data management process allows new data to be entered that refers to other data missing from the database. For example, assume that each electrophysiology recording must refer to the mouse subject used during data collection. Perhaps an experimenter attempts to insert ephys data into the database that refers to a nonexistent mouse, due to a misspelling. A system guaranteeing referential integrity, such as DataJoint, will refuse the erroneous data.

Enforcement of referential integrity does not stop with data ingest. *Deleting* data in DataJoint also deletes any dependent downstream data. Such cascading deletions are necessary to maintain referential integrity. Consider the deletion of a mouse subject without the deletion of the experimental sessions involving that mouse. A database that allows such deletion will break referential integrity, as the experimental sessions for the removed mouse depend on missing data. Any data management process that allows data to be deleted with no consideration of dependent data cannot maintain referential integrity.

Updating data already present in a database system also jeopardizes referential integrity. For this reason, the DataJoint workflow does not include updates to entities once they have been ingested into a pipeline. Allowing updates to upstream entities would break the referential integrity of any dependent data downstream. For example, permitting a user to change the name of a mouse subject would invalidate any experimental sessions that used that mouse, presuming the mouse name was part of the primary key. The proper way to change data in DataJoint is to delete the existing entities and to insert corrected ones, preserving referential integrity.

4.4.3 Group integrity

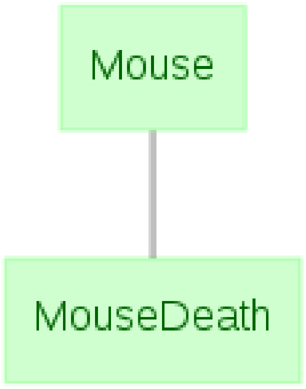
Group integrity denotes the guarantee made by the data management process that entities composed of multiple parts always appear in their complete form. Group integrity in DataJoint is formalized through *master-part* relationships. The master-part relationship has important implications for dependencies, because a downstream entity depending on a master entity set may be considered to depend on the parts as well.

4.4.4 Relationships

In DataJoint, the term **relationship** is used rather generally to describe the effects of particular configurations of *dependencies* between multiple entity sets. It is often useful to classify relationships as one-to-one, many-to-one, one-to-many, and many-to-many.

In a **one-to-one relationship**, each entity in a downstream table has exactly one corresponding entity in the upstream table. A dependency of an entity set containing the death dates of mice on an entity set describing the mice themselves would obviously be a one-to-one relationship, as in the example below.

```
@schema
class Mouse(dj.Manual):
  definition = """
  mouse_name : varchar(64)
  ---
  mouse_dob : datetime
  """

@schema
class MouseDeath(dj.Manual):
  definition = """
  -> Mouse
  ---
  death_date : datetime
  """
```



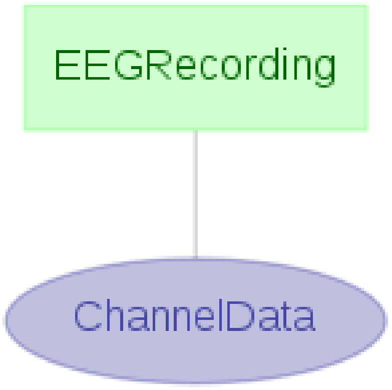
In a **one-to-many relationship**, multiple entities in a downstream table may depend on the same entity in the upstream table. The example below shows a table containing individual channel data from multi-channel recordings, representing a one-to-many relationship.

```
@schema
class EEGRecording(dj.Manual):
  definition = """
```

(continues on next page)

(continued from previous page)

```
  -> Session
  eeg_recording_id : int
  ---
  eeg_system : varchar(64)
  num_channels : int
  """

@schema
class ChannelData(dj.Imported):
  definition = """
  -> EEGRecording
  channel_idx : int
  ---
  channel_data : longblob
  """
```



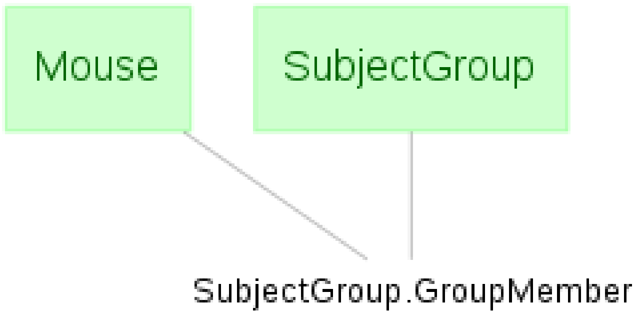
In a **many-to-one relationship**, each entity in a table is associated with multiple entities from another table. Many-to-one relationships between two tables are usually established using a separate membership table. The example below includes a table of mouse subjects, a table of subject groups, and a membership *part table* listing the subjects in each group. A many-to-one relationship exists between the `Mouse` table and the `SubjectGroup` table, with is expressed through entities in `GroupMember`.

```
@schema
class Mouse(dj.Manual):
  definition = """
  mouse_name : varchar(64)
  ---
  mouse_dob : datetime
  """

@schema
class SubjectGroup(dj.Manual):
  definition = """
  group_number : int
  ---
  group_name : varchar(64)
  """

  class GroupMember(dj.Part):
    definition = """
```

(continues on next page)

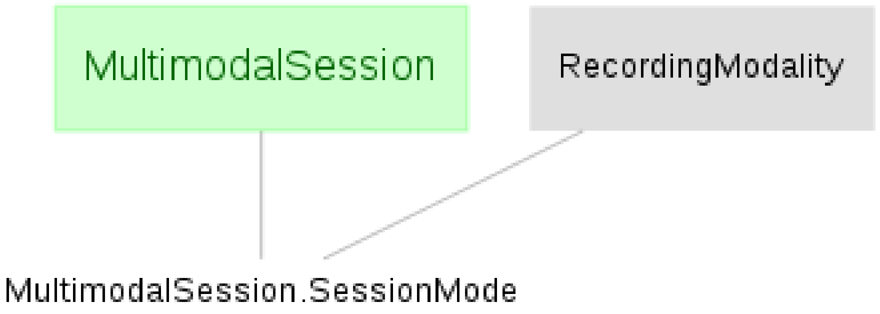
(continued from previous page)

```
    -> master
    -> Mouse
    """
```



In a **many-to-many relationship**, multiple entities in one table may each relate to multiple entities in another upstream table. Many-to-many relationships between two tables are usually established using a separate association table. Each entity in the association table links one entity from each of the two upstream tables it depends on. The below example of a many-to-many relationship contains a table of recording modalities and a table of multimodal recording sessions. Entities in a third table represent the modes used for each session.

```
@schema
class RecordingModality(dj.Lookup):
  definition = """
  modality : varchar(64)
  """

@schema
class MultimodalSession(dj.Manual):
  definition = """
  -> Session
  modes : int
  """

  class SessionMode(dj.Part):
    definition = """
    -> master
    -> RecordingModality
    """
```



The types of relationships between entity sets are expressed in the *ERD* of a schema.

CHAPTER

FIVE

DATA DEFINITION

5.1 Creating Schemas

5.1.1 Schemas

On the database server, related tables are grouped into a named collection called a **schema**. This grouping organizes the data and allows control of user access. A database server may contain multiple schemas each containing a subset of the tables. A single pipeline may comprise multiple schemas. Tables are defined within a schema, so a schema must be created before the creation of any tables.

Note: By convention, the `datajoint` package is imported as `dj`. The documentation refers to the package as `dj` throughout.

Create a new schema using the `dj.Schema` class object:

```
import datajoint as dj
schema = dj.Schema('alice_experiment')
```

This statement creates the database schema `alice_experiment` on the server.

The returned object `schema` will then serve as a decorator for DataJoint classes, as described in *Creating Tables*.

It is a common practice to have a separate Python module for each schema. Therefore, each such module has only one `dj.Schema` object defined and is usually named `schema`.

The `dj.Schema` constructor can take a number of optional parameters after the schema name.

- `context` - Dictionary for looking up foreign key references. Defaults to `None` to use local context.
- `connection` - Specifies the DataJoint connection object. Defaults to `dj.conn()`.
- `create_schema` - When `False`, the schema object will not create a schema on the database and will raise an error if one does not already exist. Defaults to `True`.
- `create_tables` - When `False`, the schema object will not create tables on the database and will raise errors when accessing missing tables. Defaults to `True`.

5.1.2 Working with existing data

See the chapter *Work with Existing Pipelines* for how to work with data in existing pipelines, including accessing a pipeline from one language when the pipeline was developed using another.

5.2 Creating Tables

5.2.1 Classes represent tables

To make it easy to work with tables in MATLAB and Python, DataJoint programs create a separate class for each table. Computer programmers refer to this concept as object-relational mapping. For example, the class `experiment.Subject` in the DataJoint client language may correspond to the table called `subject` on the database server. Users never need to see the database directly; they only interact with data in the database by creating and interacting with DataJoint classes.

Data tiers

The table class must inherit from one of the following superclasses to indicate its data tier: `dj.Lookup`, `dj.Manual`, `dj.Imported`, `dj.Computed`, or `dj.Part`. See *Data Tiers* and *Master-Part Relationship*.

5.2.2 Defining a table

To define a DataJoint table in Python:

1. Define a class inheriting from the appropriate DataJoint class: `dj.Lookup`, `dj.Manual`, `dj.Imported` or `dj.Computed`.
2. Decorate the class with the schema object (see *Creating Schemas*)
3. Define the class property `definition` to define the table heading.

For example, the following code defines the table `Person`:

```python
import datajoint as dj
schema = dj.Schema('alice_experiment')

@schema
class Person(dj.Manual):
    definition = '''
                username : varchar(20)   # unique user name
    ---
    first_name : varchar(30)
    last_name  : varchar(30)
    '''
```

The `@schema` decorator uses the class name and the data tier to check whether an appropriate table exists on the database. If a table does not already exist, the decorator creates one on the database using the definition property. The decorator attaches the information about the table to the class, and then returns the class.

The class will become usable after you define the `definition` property as described in *Table Definition*.

DataJoint classes in Python

DataJoint for Python is implemented through the use of classes providing access to the actual tables stored on the database. Since only a single table exists on the database for any class, interactions with all instances of the class are equivalent. As such, most methods can be called on the classes themselves rather than on an object, for convenience. Whether calling a DataJoint method on a class or on an instance, the result will only depend on or apply to the corresponding table. All of the basic functionality of DataJoint is built to operate on the classes themselves, even when called on an instance. For example, calling `Person.insert(...)` (on the class) and `Person.insert(...)` (on an instance) both have the identical effect of inserting data into the table on the database server. DataJoint does not prevent a user from working with instances, but the workflow is complete without the need for instantiation. It is up to the user whether to implement additional functionality as class methods or methods called on instances.

5.2.3 Valid class names

Note that in both MATLAB and Python, the class names must follow the CamelCase compound word notation:

- start with a capital letter and
- contain only alphanumerical characters (no underscores).

Examples of valid class names:

`TwoPhotonScan`, `Scan2P`, `Ephys`, `MembraneVoltage`

Invalid class names:

`Two_photon_Scan`, `twoPhotonScan`, `2PhotonScan`, `membranePotential`, `membrane_potential`

5.3 Table Definition

DataJoint models data as sets of **entities** with shared **attributes**, often visualized as tables with rows and columns. Each row represents a single entity and the values of all of its attributes. Each column represents a single attribute with a name and a datatype, applicable to entity in the table. Unlike rows in a spreadsheet, entities in DataJoint don't have names or numbers: they can only be identified by the values of their attributes. Defining a table means defining the names and datatypes of the attributes as well as the constraints to be applied to those attributes. Both MATLAB and Python use the same syntax define tables.

For example, the following code in defines the table `User`, that contains users of the database:

The table definition is contained in the `definition` property of the class.

```python
@schema
class User(dj.Manual):
    definition = """
    # database users
    username : varchar(20)   # unique user name
    ---
    first_name : varchar(30)
    last_name  : varchar(30)
    role : enum('admin', 'contributor', 'viewer')
    """
```

This defines the class `User` that creates the table in the database and provides all its data manipulation functionality.

5.3.1 Table creation on the database server

Users do not need to do anything special to have a table created in the database. Tables are created at the time of class definition. In fact, table creation on the database is one of the jobs performed by the decorator `@schema` of the class.

5.3.2 Changing the definition of an existing table

Once the table is created in the database, the definition string has no further effect. In other words, changing the definition string in the class of an existing table will not actually update the table definition. To change the table definition, one must first *drop* the existing table. This means that all the data will be lost, and the new definition will be applied to create the new empty table.

Therefore, in the initial phases of designing a DataJoint pipeline, it is common to experiment with variations of the design before populating it with substantial amounts of data.

It is possible to modify a table without dropping it. This topic is covered separately.

5.3.3 Reverse-engineering the table definition

DataJoint objects provide the `describe` method, which displays the table definition used to define the table when it was created in the database. This definition may differ from the definition string of the class if the definition string has been edited after creation of the table.

Examples

```
s = lab.User.describe()
```

5.4 Definition Syntax

The table definition consists of one or more lines. Each line can be one of the following:

- The optional first line starting with a `#` provides a description of the table's purpose. It may also be thought of as the table's long title.
- A new attribute definition in any of the following forms (see *Datatypes* for valid datatypes):

  `name : datatype` `name : datatype # comment` `name = default : datatype` `name = default : datatype # comment`

- The divider `---` (at least three hyphens) separating primary key attributes above from secondary attributes below.
- A foreign key in the format `-> ReferencedTable`. (See *Dependencies*.)

For example, the table for Persons may have the following definition:

```
# Persons in the lab
username :  varchar(16)   #  username in the database
---
full_name  : varchar(255)
start_date :  date   # date when joined the lab
```

This will define the table with attributes `username`, `full_name`, and `start_date`, in which `username` is the *primary key*.

5.4.1 Attribute names

Attribute names must be in lowercase and must start with a letter. They can only contain alphanumerical characters and underscores. The attribute name cannot exceed 64 characters.

Valid attribute names `first_name`, `two_photon_scan`, `scan_2p`, `two_photon_scan_`

Invalid attribute names `firstName`, `first name`, `2photon_scan`, `two-photon_scan`, `TwoPhotonScan`

Ideally, attribute names should be unique across all tables that are likely to be used in queries together. For example, tables often have attributes representing the start times of sessions, recordings, etc. Such attributes must be uniquely named in each table, such as `session_start_time` or `recording_start_time`.

5.4.2 Default values

Secondary attributes can be given default values. A default value will be used for an attribute if no other value is given at the time the entity is *inserted* into the table. Generally, default values are numerical values or character strings. Default values for dates must be given as strings as well, contained within quotes (with the exception of `CURRENT_TIMESTAMP`). Note that default values can only be used when inserting as a mapping. Primary key attributes cannot have default values (with the exceptions of `auto_increment` and `CURRENT_TIMESTAMP` attributes; see *Primary Key*).

An attribute with a default value of `NULL` is called a **nullable attribute**. A nullable attribute can be thought of as applying to all entities in a table but having an optional *value* that may be absent in some entities. Nullable attributes should *not* be used to indicate that an attribute is inapplicable to some entities in a table (see *Entity Normalization*). Nullable attributes should be used sparingly to indicate optional rather than inapplicable attributes that still apply to all entities in the table. `NULL` is a special literal value and does not need to be enclosed in quotes.

Here are some examples of attributes with default values:

```
failures = 0 : int
due_date = "2020-05-31" : date
additional_comments = NULL : varchar(256)
```

5.5 Data Tiers

DataJoint assigns all tables to one of the following data tiers that differentiate how the data originate.

Table 1: Table tiers

Tier	Superclass	Description
Lookup	`dj.Lookup`	Small tables containing general facts and settings of the data pipeline; not specific to any experiment or dataset.
Manual	`dj.Manual`	Data entered from outside the pipeline, either by hand or with external helper scripts.
Imported	`dj.Imported`	Data ingested automatically inside the pipeline but requiring access to data outside the pipeline.
Computed	`dj.Computed`	Data computed automatically entirely inside the pipeline.

Table data tiers indicate to database administrators how valuable the data are. Manual data are the most valuable, as re-entry may be tedious or impossible. Computed data are safe to delete, as the data can always be recomputed from within DataJoint. Imported data are safer than manual data but less safe than computed data because of dependency on external data sources. With these considerations, database administrators may opt not to back up computed data, for example, or to back up imported data less frequently than manual data.

The data tier of a table is specified by the superclass of its class. For example, the User class in *Table Definition* uses the `dj.Manual` superclass. Therefore, the corresponding User table on the database would be of the Manual tier. Furthermore, the classes for **imported** and **computed** tables have additional capabilities for automated processing as described in *Auto-populate*.

5.5.1 Internal conventions for naming tables

On the server side, DataJoint uses a naming scheme to generate a table name corresponding to a given class. The naming scheme includes prefixes specifying each table's data tier.

First, the name of the class is converted from `CamelCase` to `snake_case` (separation by underscores). Then the name is prefixed according to the data tier.

- `Manual` tables have no prefix.
- `Lookup` tables are prefixed with `#`.
- `Imported` tables are prefixed with `_`, a single underscore.
- `Computed` tables are prefixed with `__`, two underscores.

For example:

The table for the class `StructuralScan` subclassing `dj.Manual` will be named `structural_scan`.

The table for the class `SpatialFilter` subclassing `dj.Lookup` will be named `#spatial_filter`.

Again, the internal table names including prefixes are used only on the server side. These are never visible to the user, and DataJoint users do not need to know these conventions However, database administrators may use these naming patterns to set backup policies or to restrict access based on data tiers.

5.5.2 Part tables

Part tables do not have their own tier. Instead, they share the same tier as their master table. The prefix for part tables also differs from the other tiers. They are prefixed by the name of their master table, separated by two underscores.

For example, the table for the class `Channel(dj.Part)` with the master `Ephys(dj.Imported)` will be named `_ephys__channel`.

5.6 Datatypes

DataJoint supports the following datatypes. To conserve database resources, use the smallest and most restrictive datatype sufficient for your data. This also ensures that only valid data are entered into the pipeline.

5.6.1 Most common datatypes

- `tinyint` : an 8-bit integer number, ranging from -128 to 127.
- `tinyint unsigned` : an 8-bit positive integer number, ranging from 0 to 255.
- `smallint` : a 16-bit integer number, ranging from -32,768 to 32,767.
- `smallint unsigned` : a 16-bit positive integer, ranging from 0 to 65,535.
- `int` : a 32-bit integer number, ranging from -2,147,483,648 to 2,147,483,647.
- `int unsigned` : a 32-bit positive integer, ranging from 0 to 4,294,967,295.
- `enum` : one of several explicitly enumerated values specified as strings. Use this datatype instead of text strings to avoid spelling variations and to save storage space. For example, the datatype for an anesthesia attribute could be `enum("urethane", "isoflurane", "fentanyl")` . Do not use enums in primary keys due to the difficulty of changing their definitions consistently in multiple tables.
- `date` : date as `'YYYY-MM-DD'` .
- `time` : time as `'HH:MM:SS'` .
- `datetime` : Date and time to the second as `'YYYY-MM-DD HH:MM:SS'`
- `timestamp` : Date and time to the second as `'YYYY-MM-DD HH:MM:SS'` . The default value may be set to `CURRENT_TIMESTAMP` . Unlike `datetime` , a `timestamp` value will be adjusted to the local time zone.
- `char(N)` : a character string up to *N* characters (but always takes the entire *N* bytes to store).
- `varchar(N)` : a text string of arbitrary length up to *N* characters that takes *M+1* or *M+2* bytes of storage, where *M* is the actual length of each stored string.
- `float` : a single-precision floating-point number. Takes 4 bytes. Single precision is sufficient for many measurements.
- `double` : a double-precision floating-point number. Takes 8 bytes. Because equality comparisons are error-prone, neither `float` nor `double` should be used in primary keys.
- `decimal(N,F)` : a fixed-point number with *N* total decimal digits and *F* fractional digits. This datatype is well suited to represent numbers whose magnitude is well defined and does not warrant the use of floating-point representation or requires precise decimal representations (e.g. dollars and cents). Because of its well-defined precision, `decimal` values can be used in equality comparison and be included in primary keys.
- `longblob` : arbitrary numeric array (e.g. matrix, image, structure), up to 4 GiB in size. Numeric arrays are compatible between MATLAB and Python (NumPy). The `longblob` and other `blob` datatypes can be configured to store data *externally* by using the `blob@store` syntax.

5.6.2 Less common (but supported) datatypes

- `decimal(N,F) unsigned` : same as `decimal` , but limited to nonnegative values.
- `mediumint` a 24-bit integer number, ranging from -8,388,608 to 8,388,607.
- `mediumint unsigned` : a 24-bit positive integer, ranging from 0 to 16,777,216.
- `mediumblob` : arbitrary numeric array, up to 16 MiB
- `blob` : arbitrary numeric array, up to 64 KiB

- `tinyblob` : arbitrary numeric array, up to 256 bytes (actually smaller due to header info).

5.6.3 Special DataJoint-only datatypes

These types abstract certain kinds of non-database data to facillitate use together with DataJoint.

- `attach` : a *file attachment* similar to email attachments facillitating sending/receiving an opaque data file to/from a DataJoint pipeline.
- `filepath@store` : a *filepath* used to link non-DataJoint managed files into a DataJoint pipeline.

5.6.4 Datatypes not (yet) supported

- `binary`
- `text`
- `longtext`
- `bit`

For additional information about these datatypes, see http://dev.mysql.com/doc/refman/5.6/en/data-types.html

5.7 External Data

5.7.1 File Attachment Datatype

Configuration & Usage

Corresponding to issue #480, the `attach` attribute type allows users to *attach* files into DataJoint schemas as DataJoint-managed files. This is in contrast to traditional *blobs* which are encodings of programming language data structures such as arrays.

The functionality is modeled after email attachments, where users *attach* a file along with a message and message recipients have access to a copy of that file upon retrieval of the message.

For DataJoint `attach` attributes, DataJoint will copy the input file into a DataJoint store, hash the file contents, and track the input file name. Subsequent `fetch` operations will transfer a copy of the file to the local directory of the Python process and return a pointer to it's location for subsequent client usage. This allows arbitrary files to be *uploaded* or *attached* to a DataJoint schema for later use in processing. File integrity is preserved by checksum comparison against the attachment data and verifying the contents during retrieval.

For example, given a `localattach` store:

```
dj.config['stores'] = {
  'localattach': {
    'protocol': 'file',
    'location': '/data/attach'
  }
}
```

A `ScanAttachment` table can be created:

```
@schema
class ScanAttachment(dj.Manual):
    definition = """
    -> Session
    ---
    scan_image:    attach@localattach  # attached image scans
    """
```

Files can be added using an insert pointing to the source file:

```
>>> ScanAttachment.insert1((0, '/input/image0.tif'))
```

And then retrieved to the current directory using `fetch`:

```
>>> s0 = (ScanAttachment & {'session_id': 0}).fetch1()
>>> s0
{'session_id': 0, 'scan_image': './image0.tif'}
>>> fh = open(s0['scan_image'], 'rb')
>>> fh
<_io.BufferedReader name='./image0.tif')
```

5.7.2 Filepath Datatype

Note: Filepath Datatype is available as a preview feature in DataJoint Python v0.12. This means that the feature is required to be explicitly enabled. To do so, make sure to set the environment variable `FILEPATH_FEATURE_SWITCH=TRUE` prior to use.

Configuration & Usage

Corresponding to issue #481, the `filepath` attribute type links DataJoint records to files already managed outside of DataJoint. This can aid in sharing data with other systems such as allowing an image viewer application to directly use files from a DataJoint pipeline, or to allow downstream tables to reference data which reside outside of DataJoint pipelines.

To define a table using the `filepath` datatype, an existing DataJoint *store* should be created and then referenced in the new table definition. For example, given a simple store:

```
dj.config['stores'] = {
  'data': {
    'protocol': 'file',
    'location': '/data',
    'stage': '/data'
  }
}
```

we can define an `ScanImages` table as follows:

```
@schema
class ScanImages(dj.Manual):
    definition = """
    -> Session
    image_id:    int
```

(continues on next page)

(continued from previous page)

```
    ---
    image_path:  filepath@data
    """
```

This table can now be used for tracking paths within the `/data` local directory. For example:

```
>>> ScanImages.insert1((0, 0, '/data/images/image_0.tif'))
>>> (ScanImages() & {'session_id': 0}).fetch1(as_dict=True)
{'session_id': 0, 'image_id': 0, 'image_path': '/data/images/image_0.tif'}
```

As can be seen from the example, unlike *blob* records, file paths are managed as path locations to the underlying file.

Integrity Notes

Unlike other data in DataJoint, data in `filepath` records are deliberately intended for shared use outside of DataJoint. To help ensure integrity of `filepath` records, DataJoint will record a checksum of the file data on `insert`, and will verify this checksum on `fetch`. However, since the underlying file data may be shared with other applications, special care should be taken to ensure records stored in `filepath` attributes are not modified outside of the pipeline, or, if they are, that records in the pipeline are updated accordingly. A safe method of changing `filepath` data is as follows:

1) Delete the `filepath` database record.

   This will ensure that any downstream records in the pipeline depending on the `filepath` record are purged from the database.

2) Modify `filepath` data.

3) Re-insert corresponding the `filepath` record.

   This will add the record back to DataJoint with an updated file checksum.

4) Compute any downstream dependencies, if needed.

   This will ensure that downstream results dependent on the `filepath` record are updated to reflect the newer `filepath` contents.

5.8 Primary Key

5.8.1 Primary keys in DataJoint

Entities in tables are neither named nor numbered. DataJoint does not answer questions of the type "What is the 10th element of this table?" Instead, entities are distinguished by the values of their attributes. Furthermore, the entire entity is not required for identification. In each table, a subset of its attributes are designated to be the **primary key**. Attributes in the primary key alone are sufficient to differentiate any entity from any other within the table.

Each table must have exactly one primary key: a subset of its attributes that uniquely identify each entity in the table. The database uses the primary key to prevent duplicate entries, to relate data across tables, and to accelerate data queries. The choice of the primary key will determine how you identify entities. Therefore, make the primary key **short**, **expressive**, and **persistent**.

For example, mice in our lab are assigned unique IDs. The mouse ID number `animal_id` of type `smallint` can serve as the primary key for the table `Mice`. An experiment performed on a mouse may be identified in the table `Experiments` by two attributes: `animal_id` and `experiment_number`.

DataJoint takes the concept of primary keys somewhat more seriously than other models and query languages. Even **table expressions**, i.e. those tables produced through operations on other tables, have a well-defined primary key. All operators on tables are designed in such a way that the results always have a well-defined primary key.

In all representations of tables in DataJoint, the primary key attributes are always listed before other attributes and highlighted for emphasis (e.g. in a **bold** font or marked with an asterisk *)

5.8.2 Defining a primary key

In table declarations, the primary key attributes always come first and are separated from the other attributes with a line containing at least three hyphens. For example, the following is the definition of a table containing database users where `username` is the primary key.

```
# database users
username : varchar(20)   # unique user name
---
first_name : varchar(30)
last_name  : varchar(30)
role : enum('admin', 'contributor', 'viewer')
```

5.8.3 Entity integrity

The primary key defines and enforces the desired property of databases known as *entity integrity*. **Entity integrity** ensures that there is a one-to-one and unambiguous mapping between real-world entities and their representations in the database system. The data management process must prevent any duplication or misidentification of entities.

To enforce entity integrity, DataJoint implements several rules: * Every table must have a primary key. * Primary key attributes cannot have default values (with the exception of `auto_increment` and `CURRENT_TIMESTAMP`; see below). * Operators on tables are defined with respect to the primary key and preserve a primary key in their results.

5.8.4 Datatypes in primary keys

All integer types, dates, timestamps, and short character strings make good primary key attributes. Character strings are somewhat less suitable because they can be long and because they may have invisible trailing spaces. Floating-point numbers should be avoided because rounding errors may lead to misidentification of entities. Enums are okay as long as they do not need to be modified after *dependencies* are already created referencing the table. Finally, DataJoint does not support blob types in primary keys.

The primary key may be composite, i.e. comprising several attributes. In DataJoint, hierarchical designs often produce tables whose primary keys comprise many attributes.

5.8.5 Choosing primary key attributes

A primary key comprising real-world attributes is a good choice when such real-world attributes are already properly and permanently assigned. Whatever characteristics are used to uniquely identify the actual entities can be used to identify their representations in the database.

If there are no attributes that could readily serve as a primary key, an artificial attribute may be created solely for the purpose of distinguishing entities. In such cases, the primary key created for management in the database must also be used to uniquely identify the entities themselves. If the primary key resides only in the database while entities remain indistinguishable in the real world, then the process cannot ensure entity integrity. When a primary key is created as part of data management rather than based on real-world attributes, an institutional process must ensure the uniqueness and permanence of such an identifier.

For example, the U.S. government assigns every worker an identifying attribute, the social security number. However, the government must go to great lengths to ensure that this primary key is assigned exactly once, by checking against other less convenient candidate keys (i.e. the combination of name, parents' names, date of birth, place of birth, etc.). Just like the SSN, well managed primary keys tend to get institutionalized and find multiple uses.

Your lab must maintain a system for uniquely identifying important entities. For example, experiment subjects and experiment protocols must have unique IDs. Use these as the primary keys in the corresponding tables in your DataJoint databases.

Using hashes as primary keys

Some tables include too many attributes in their primary keys. For example, the stimulus condition in a psychophysics experiment may have a dozen parameters such that a change in any one of them makes a different valid stimulus condition. In such a case, all the attributes would need to be included in the primary key to ensure entity integrity. However, long primary keys make it difficult to reference individual entities. To be most useful, primary keys need to be relatively short.

This problem is effectively solved through the use of a hash of all the identifying attributes as the primary key. For example, MD5 or SHA-1 hash algorithms can be used for this purpose. To keep their representations human-readable, they may be encoded in base-64 ASCII. For example, the 128-bit MD5 hash can be represented by 21 base-64 ASCII characters, but for many applications, taking the first 8 to 12 characters is sufficient to avoid collisions.

`auto_increment`

Some entities are created by the very action of being entered into the database. The action of entering them into the database gives them their identity. It is impossible to duplicate them since entering the same thing twice still means creating two distinct entities.

In such cases, the use of an auto-incremented primary key is warranted. These are declared by adding the word `auto_increment` after the data type in the declaration. The datatype must be an integer. Then the database will assign incrementing numbers at each insert.

The example definition below defines an auto-incremented primary key

```
# log entries
entry_id  :  smallint auto_increment
---
entry_text :  varchar(4000)
entry_time = CURRENT_TIMESTAMP : timestamp(3)  # automatic timestamp with millisecond precision
```

DataJoint passes `auto_increment` behavior to the underlying MySQL and therefore it has the same limitation: it can only be used for tables with a single attribute in the primary key.

If you need to auto-increment an attribute in a composite primary key, you will need to do so programmatically within a transaction to avoid collisions.

For example, let's say that you want to auto-increment `scan_idx` in a table called `Scan` whose primary key is `(animal_id, session, scan_idx)`. You must already have the values for `animal_id` and `session` in the dictionary `key`. Then you can do the following:

```
U().aggr(Scan & key, next='max(scan_idx)+1')

# or

Session.aggr(Scan, next='max(scan_idx)+1') & key
```

Note that the first option uses a *universal set*.

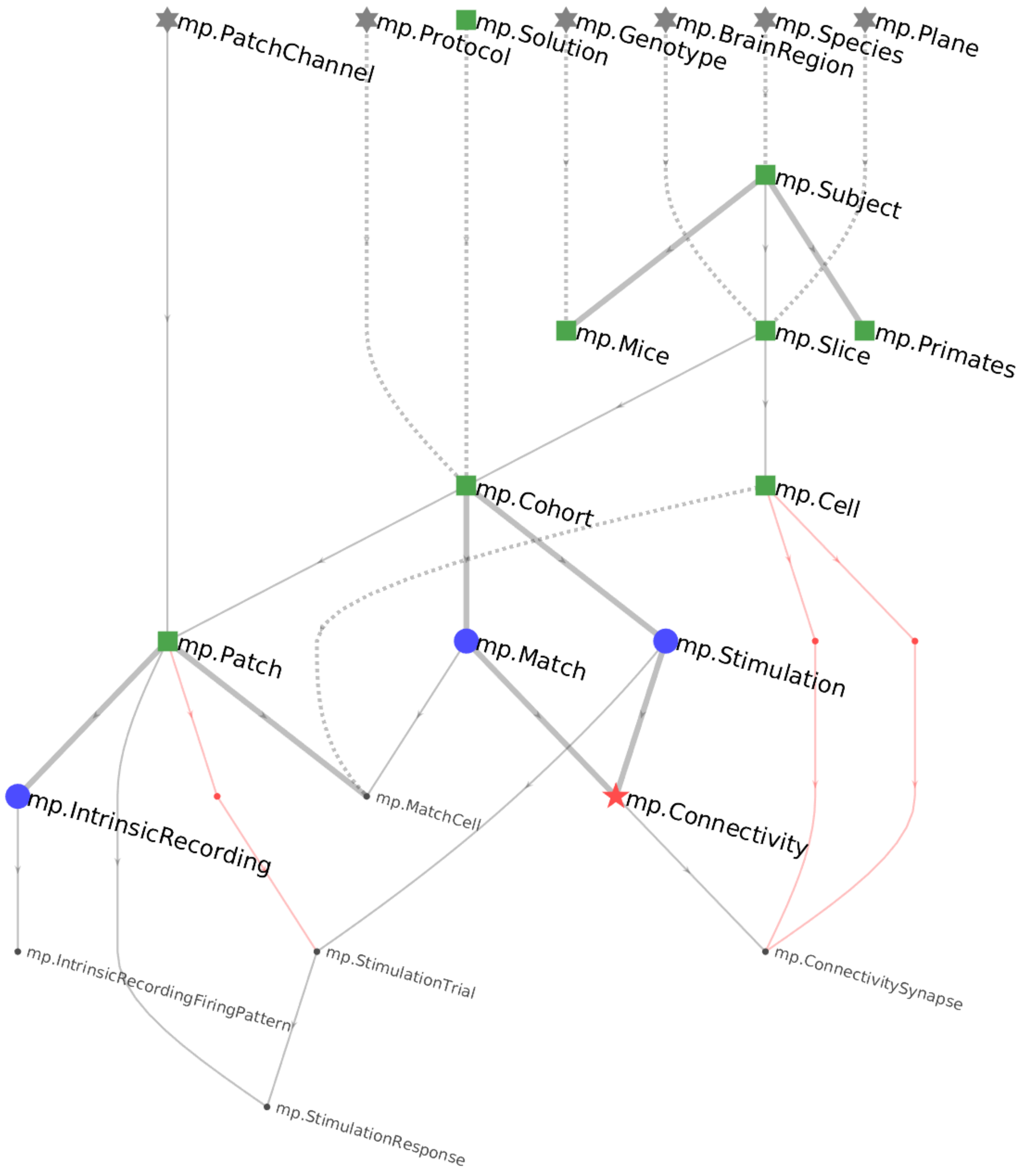
5.9 Dependencies

5.9.1 Understanding dependencies

A schema contains collections of tables of related data. Accordingly, entities in one table often derive some of their meaning or context from entities in other tables. A **foreign key** defines a **dependency** of entities in one table on entities in another within a schema. In more complex designs, dependencies can even exist between entities in tables from different schemas. Dependencies play a functional role in DataJoint and do not simply label the structure of a pipeline. Dependencies provide entities in one table with access to data in another table and establish certain constraints on entities containing a foreign key.

A DataJoint pipeline, including the dependency relationships established by foreign keys, can be visualized as a graph with nodes and edges. The diagram of such a graph is called the **entity relationship diagram** or *ERD*. The nodes of the graph are tables and the edges connecting them are foreign keys. The edges are directed and the overall graph is a **directed acyclic graph**, a graph with no loops.

For example, the ERD below is the pipeline for multipatching experiments



The graph defines the direction of the workflow. The tables at the top of the flow need to be populated first, followed by those tables one step below and so forth until the last table is populated at the bottom of the pipeline. The top of the pipeline tends to be dominated by lookup tables (gray stars) and manual tables (green squares). The middle has many imported tables (blue triangles), and the bottom has computed tables (red stars).

5.9.2 Defining a dependency

Foreign keys are defined with arrows `->` in the *table definition*, pointing to another table.

A foreign key may be defined as part of the *Primary Key*.

In the ERD, foreign keys from the primary key are shown as solid lines. This means that the primary key of the referenced table becomes part of the primary key of the new table. A foreign key outside the primary key is indicated by dashed line in the ERD.

For example, the following definition for the table `mp.Slice` has three foreign keys, including one within the primary key.

```
# brain slice
-> mp.Subject
slice_id        : smallint       # slice number within subject
---
-> mp.BrainRegion
-> mp.Plane
slice_date        : date                 # date of the slicing (not patching)
thickness         : smallint unsigned    # slice thickness in microns
experimenter      : varchar(20)          # person who performed this experiment
```

You can examine the resulting table heading with

```
mp.BrainSlice.heading
```

The heading of `mp.Slice` may look something like

```
subject_id      : char(8)        # experiment subject id
slice_id        : smallint       # slice number within subject
---
brain_region        : varchar(12)       # abbreviated name for brain region
plane               : varchar(12)       # plane of section
slice_date          : date              # date of the slicing (not patching)
thickness           : smallint unsigned # slice thickness in microns
experimenter        : varchar(20)       # person who performed this experiment
```

This displayed heading reflects the actual attributes in the table. The foreign keys have been replaced by the primary key attributes of the referenced tables, including their data types and comments.

5.9.3 How dependencies work

The foreign key `-> A` in the definition of table `B` has the following effects:

1. The primary key attributes of `A` are made part of `B`'s definition.
2. A referential constraint is created in `B` with reference to `A`.
3. If one does not already exist, an index is created to speed up searches in `B` for matches to `A`. (The reverse search is already fast because it uses the primary key of `A`.)

A referential constraint means that an entity in `B` cannot exist without a matching entity in `A`. **Matching** means attributes in `B` that correspond to the primary key of `A` must have the same values. An attempt to insert an entity into `B` that does not have a matching counterpart in `A` will fail. Conversely, deleting an entity from `A` that has matching entities in `B` will result in the deletion of those matching entities and so forth, recursively, downstream in the pipeline.

When `B` references `A` with a foreign key, one can say that `B` **depends** on `A`. In DataJoint terms, `B` is the **dependent table** and `A` is the **referenced table** with respect to the foreign key from `B` to `A`.

Note: Note to those already familiar with the theory of relational databases: The usage of the words "depends" and "dependency" here should not be confused with the unrelated concept of *functional dependencies* that is used to define normal forms.

5.9.4 Referential integrity

Dependencies enforce the desired property of databases known as **referential integrity**. Referential integrity is the guarantee made by the data management process that related data across the database remain present, correctly associated, and mutually consistent. Guaranteeing referential integrity means enforcing the constraint that no entity can exist in the database without all the other entities on which it depends. An entity in table `B` depends on an entity in table `A` when they belong to them or are computed from them.

5.9.5 Dependencies with renamed attributes

In most cases, a dependency includes the primary key attributes of the referenced table as they appear in its table definition. Sometimes it can be helpful to choose a new name for a foreign key attribute that better fits the context of the dependent table. DataJoint provides the following *projection* syntax to rename the primary key attributes when they are included in the new table.

The dependency

```
->  Table.project(new_attr='old_attr')
```

renames the primary key attribute `old_attr` of `Table` as `new_attr` before integrating it into the table definition. Any additional primary key attributes will retain their original names. For example, the table `Experiment` may depend on table `User` but rename the `user` attribute into `operator` as follows:

```
-> User.proj(operator='user')
```

In the above example, an entity in the dependent table depends on exactly one entity in the referenced table. Sometimes entities may depend on multiple entities from the same table. Such a design requires a way to distinguish between dependent attributes having the same name in the reference table. For example, a table for `Synapse` may reference the table `Cell` twice as `presynaptic` and `postsynaptic`. The table definition may appear as

```
# synapse between two cells
-> Cell.proj(presynaptic='cell_id')
-> Cell.proj(postsynaptic='cell_id')
---
connection_strength : double  # (pA) peak synaptic current
```

If the primary key of `Cell` is (`animal_id`, `slice_id`, `cell_id`), then the primary key of `Synapse` resulting from the above definition will be (`animal_id`, `slice_id`, `presynaptic`, `postsynaptic`). Projection always returns all of the primary key attributes of a table, so `animal_id` and `slice_id` are included, with their original names.

Note that the design of the `Synapse` table above imposes the constraint that the synapse can only be found between cells in the same animal and in the same slice.

Allowing representation of synapses between cells from different slices requires the renamimg of `slice_id` as well: .. code-block:: text

> # synapse between two cells -> Cell(presynaptic_slice='slice_id', presynaptic_cell='cell_id') -> Cell(postsynaptic_slice='slice_id', postsynaptic_cell='cell_id') — connection_strength : double # (pA) peak synaptic current

In this case, the primary key of `Synapse` will be (`animal_id` , `presynaptic_slice` , `presynaptic_cell` , `postsynaptic_slice` , `postsynaptic_cell`). This primary key still imposes the constraint that synapses can only form between cells within the same animal but now allows connecting cells across different slices.

In the ERD, renamed foreign keys are shown as red lines with an additional dot node in the middle to indicate that a renaming took place.

5.9.6 Foreign key options

Note: Foreign key options are currently in development.

Foreign keys allow the additional options `nullable` and `unique` , which can be inserted in square brackets following the arrow.

For example, in the following table definition

```
rig_id  : char(4)   # experimental rig
---
-> Person
```

each rig belongs to a person, but the table definition does not prevent one person owning multiple rigs. With the `unique` option, a person may only appear once in the entire table, which means that no one person can own more than one rig.

```
rig_id  : char(4)   # experimental rig
---
-> [unique] Person
```

With the `nullable` option, a rig may not belong to anyone, in which case the foreign key attributes for `Person` are set to `NULL` :

```
rig_id  : char(4)   # experimental rig
---
-> [nullable] Person
```

Finally with both *unique* and *nullable*, a rig may or may not be owned by anyone and each person may own up to one rig.

```
rig_id  : char(4)   # experimental rig
---
-> [unique, nullable] Person
```

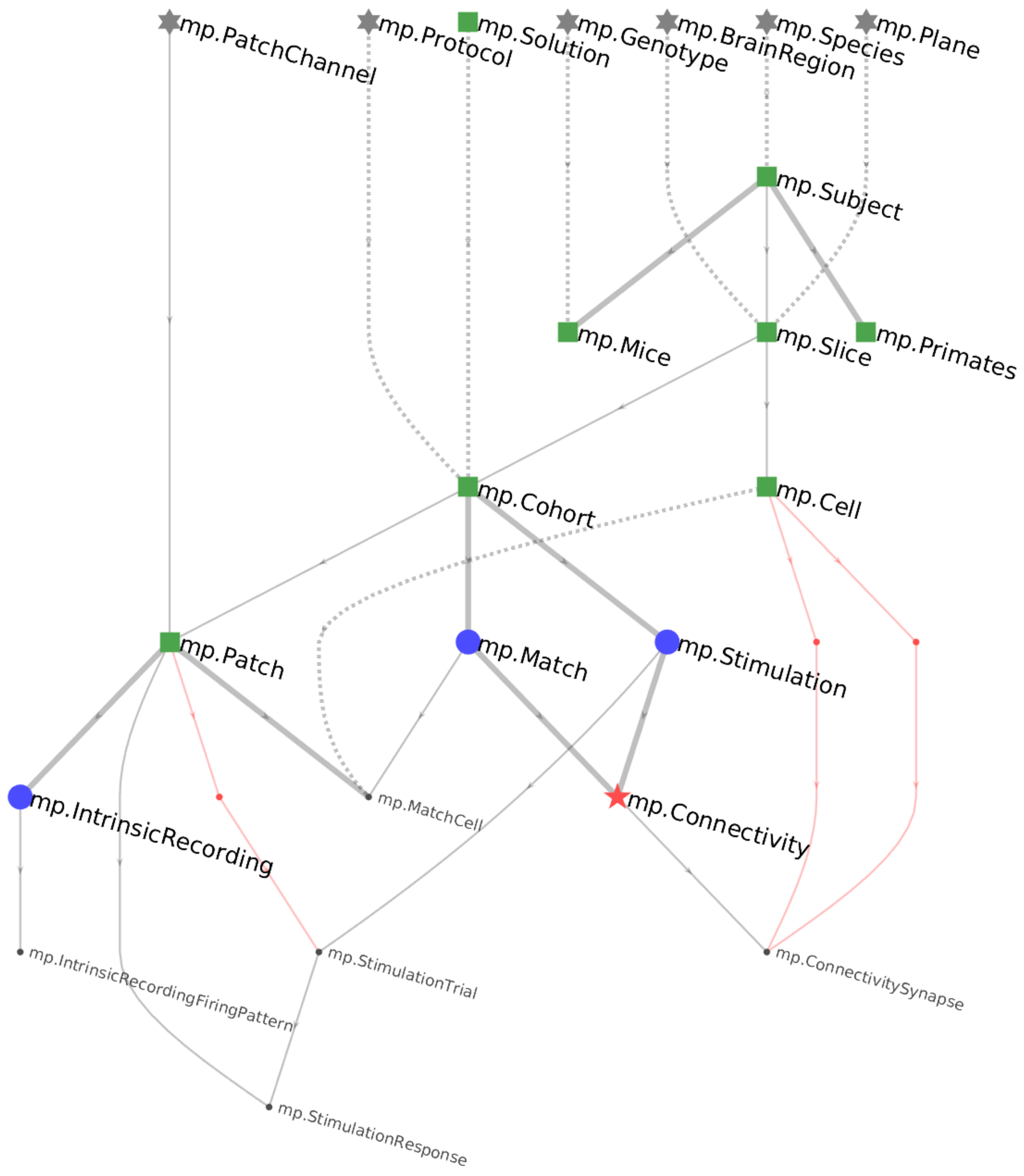
Foreign keys made from the primary key cannot be nullable but may be unique.

5.10 ERD

ERD stands for **entity relationship diagram**. Objects of type `dj.ERD` allow visualizing portions of the data pipeline in graphical form. Tables are depicted as nodes and *dependencies* as directed edges between them. The *draw* method plots the graph.

5.10.1 Diagram notation

Consider the following ERD



DataJoint uses the following conventions:

- Tables are indicated as nodes in the graph. The corresponding class name is indicated by each node.
- *Data tiers* are indicated as colors and symbols: Lookup=gray asterisk, Manual=green square, Imported=blue circle, Computed=red star, Part=black dot. The names of *part tables* are indicated in a smaller font.
- *Dependencies* are indicated as edges in the graph and always directed downward, forming a **directed acyclic**

graph.

- Foreign keys contained within the primary key are indicated as solid lines. This means that the referenced table becomes part of the primary key of the dependent table.
- Foreign keys that are outside the primary key are indicated by dashed lines.
- If the primary key of the dependent table has no other attributes besides the foreign key, the foreign key is a thick solid line, indicating a 1:{0,1} relationship.
- Foreign keys made without renaming the foreign key attributes are in black whereas foreign keys that rename the attributes are indicated in red.

5.10.2 Diagramming an entire schema

To plot the ERD for an entire schema, an ERD object can be initialized with the schema object (which is normally used to decorate table objects)

```
import datajoint as dj
schema = dj.Schema('my_database')
dj.ERD(schema).draw()
```

or alternatively an object that has the schema object as an attribute, such as the module defining a schema:

```
import datajoint as dj
import seq    # import the sequence module defining the seq database
dj.ERD(seq).draw()    # draw the ERD
```

Note that calling the `.draw()` method is not necessary when working in a Jupyter notebook. The preferred workflow is to simply let the object display itself, for example by writing `dj.ERD(seq)`. The ERD will then render in the notebook using its `_repr_html_` method. An ERD displayed without `.draw()` will be rendered as an SVG, and hovering the mouse over a table will reveal a compact version of the output of the `.describe()` method.

Initializing with a single table

A *dj.ERD* object can be initialized with a single table.

```
dj.ERD(seq.Genome).draw()
```

A single node makes a rather boring graph but ERDs can be added together or subtracted from each other using graph algebra.

Adding ERDs together

However two graphs can be added, resulting in new graph containing the union of the sets of nodes from the two original graphs. The corresponding foreign keys will be automatically

```
# plot the ERD with tables Genome and Species from module seq.
(dj.ERD(seq.Genome) + dj.ERD(seq.Species)).draw()
```

Expanding ERDs upstream and downstream

Adding a number to an ERD object adds nodes downstream in the pipeline while subtracting a number from ERD object adds nodes upstream in the pipeline.

Examples:

```
# Plot all the tables directly downstream from ``seq.Genome``:
(dj.ERD(seq.Genome)+1).draw()
```

```
# Plot all the tables directly upstream from ``seq.Genome``:
(dj.ERD(seq.Genome)-1).draw()
```

```
# Plot the local neighborhood of ``seq.Genome``
(dj.ERD(seq.Genome)+1-1+1-1).draw()
```

5.11 Manual Tables

Manual tables are populated during experiments through a variety of interfaces. Not all manual information is entered by typing. Automated software can enter it directly into the database. What makes a manual table manual is that it does not perform any computations within the DataJoint pipeline.

The following code defines three manual tables `Animal`, `Session`, and `Scan`:

```
@schema
class Animal(dj.Manual):
    definition = """
    # information about animal
    animal_id : int  # animal id assigned by the lab
    ---
    -> Species
    date_of_birth=null : date  # YYYY-MM-DD optional
    sex='' : enum('M', 'F', '')   # leave empty if unspecified
    """

@schema
class Session(dj.Manual):
    definition = """
    # Experiment Session
    -> Animal
    session  : smallint  # session number for the animal
    ---
    session_date : date  # YYYY-MM-DD
    -> User
    -> Anesthesia
    -> Rig
    """

@schema
class Scan(dj.Manual):
    definition = """
    # Two-photon imaging scan
    -> Session
    scan : smallint  # scan number within the session
    ---
```

(continues on next page)

(continued from previous page)

```
    -> Lens
    laser_wavelength : decimal(5,1)  # um
    laser_power      : decimal(4,1)  # mW
    """
```

5.12 Lookup Tables

Lookup tables contain basic facts that are not specific to an experiment and are fairly persistent. Their contents are typically small. In GUIs, lookup tables are often used for drop-down menus or radio buttons. In computed tables, they are often used to specify alternative methods for computations. Lookup tables are commonly populated from their `contents` property. In an *ERD* they are shown in gray. The decision of which tables are lookup tables and which are manual can be somewhat arbitrary.

The table below is declared as a lookup table with its contents property provided to generate entities.

```
@schema
class User(dj.Lookup):
    definition = """
    # users in the lab
    username : varchar(20)   # user in the lab
    ---
    first_name  : varchar(20)   # user first name
    last_name   : varchar(20)   # user last name
    """
    contents = [
        ['cajal', 'Santiago', 'Cajal'],
        ['hubel', 'David', 'Hubel'],
        ['wiesel', 'Torsten', 'Wiesel']
]
```

5.13 Drop

The `drop` method completely removes a table from the database, including its definition. It also removes all dependent tables, recursively. DataJoint will first display the tables being dropped and the number of entities in each before prompting the user for confirmation to proceed.

The `drop` method is often used during initial design to allow altered table definitions to take effect.

```
# drop the Person table from its schema
Person.drop()
```

5.13.1 Dropping part tables

A *part table* is usually removed as a consequence of calling `drop` on its master table.

A *part table* is usually removed as a consequence of calling `drop` on its master table. To enforce this workflow, calling `drop` directly on a part table produces an error. In some cases, it may be necessary to override this behavior. To remove a part table without removing its master, use the argument `force=True`.

CHAPTER

SIX

WORK WITH EXISTING PIPELINES

6.1 Loading Classes

This section describes how to work with database schemas without access to the original code that generated the schema. These situations often arise when the database is created by another user who has not shared the generating code yet or when the database schema is created from a programming language other than Python.

```
import datajoint as dj
```

6.1.1 Working with schemas and their modules

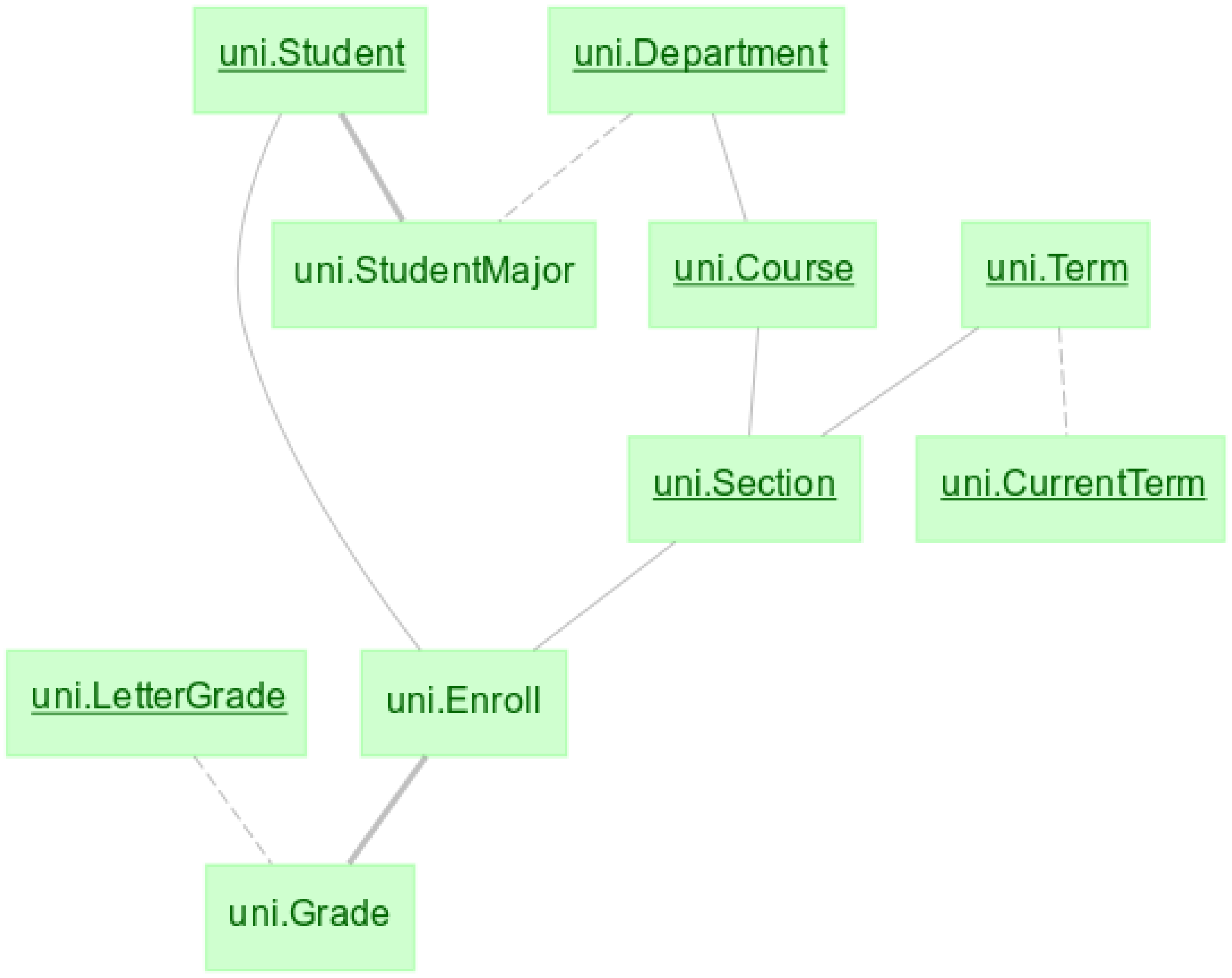
Typically a DataJoint schema is created as a dedicated Python module. This module defines a schema object that is used to link classes declared in the module to tables in the database schema. As an example, examine the university module: university.py.

You may then import the module to interact with its tables:

```
import university as uni
```

Connecting dimitri@localhost:3306

```
dj.Diagram(uni)
```



Note that dj.Diagram can extract the diagram from a schema object or from a Python module containing its schema object, lending further support to the convention of one-to-one correspondence between database schemas and Python modules in a DataJoint project:

`dj.Diagram(uni)`

is equvalent to

`dj.Diagram(uni.schema)`

```
# students without majors
uni.Student - uni.StudentMajor
```

student_id university-wide ID number	**first_name**	**last_name**	**sex**	**date_of_birth**	**home_address** mailing street address	**home_city** mailing address	**home_state** US state acronym: e.g. OH	**home_zip** zipcode e.g. 93979-4979	**home_phone** e.g. 414.657.6883x0
1003	Jonathan	Wilson	M	2002-10-19	91101 Summer Park	Port Jacqueline	VT	75616	982.251.1567x2
1005	Richard	Lopez	M	1993-10-16	86210 Brooks Station	West David	IL	96184	+1-385-481-6761x1575
1006	Laura	Hammond	F	1984-12-03	0346 Shannon Motorway	East David	ME	22113	(189)406-2652x

…

Total: 751

6.1.2 Spawning missing classes

Now imagine that you do not have access to `university.py` or you do not have its latest version. You can still connect to the database schema but you will not have classes declared to interact with it.

So let's start over in this scenario.

You can may use the `dj.list_schemas` function (new in DataJoint 0.12.0) to list the names of database schemas available to you.

```
import datajoint as dj
dj.list_schemas()
```

Connecting dimitri@localhost:3306

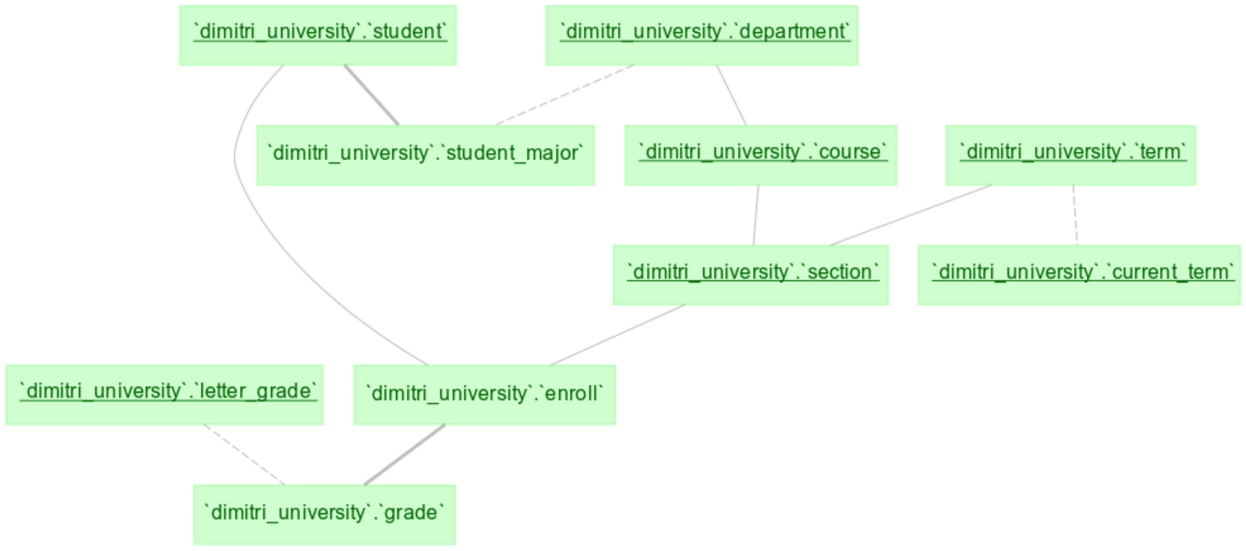
['dimitri_alter','dimitri_attach','dimitri_blob','dimitri_blobs', 'dimitri_nphoton','dimitri_schema','dimitri_university','dimitri_uuid', 'university']

Just as with a new schema, we start by creating a schema object to connect to the chosen database schema:

```
schema = dj.Schema('dimitri_university')
```

If the schema already exists, dj.Schema is initialized as usual and you may plot the schema diagram. But instead of seeing class names, you will see the raw table names as they appear in the database.

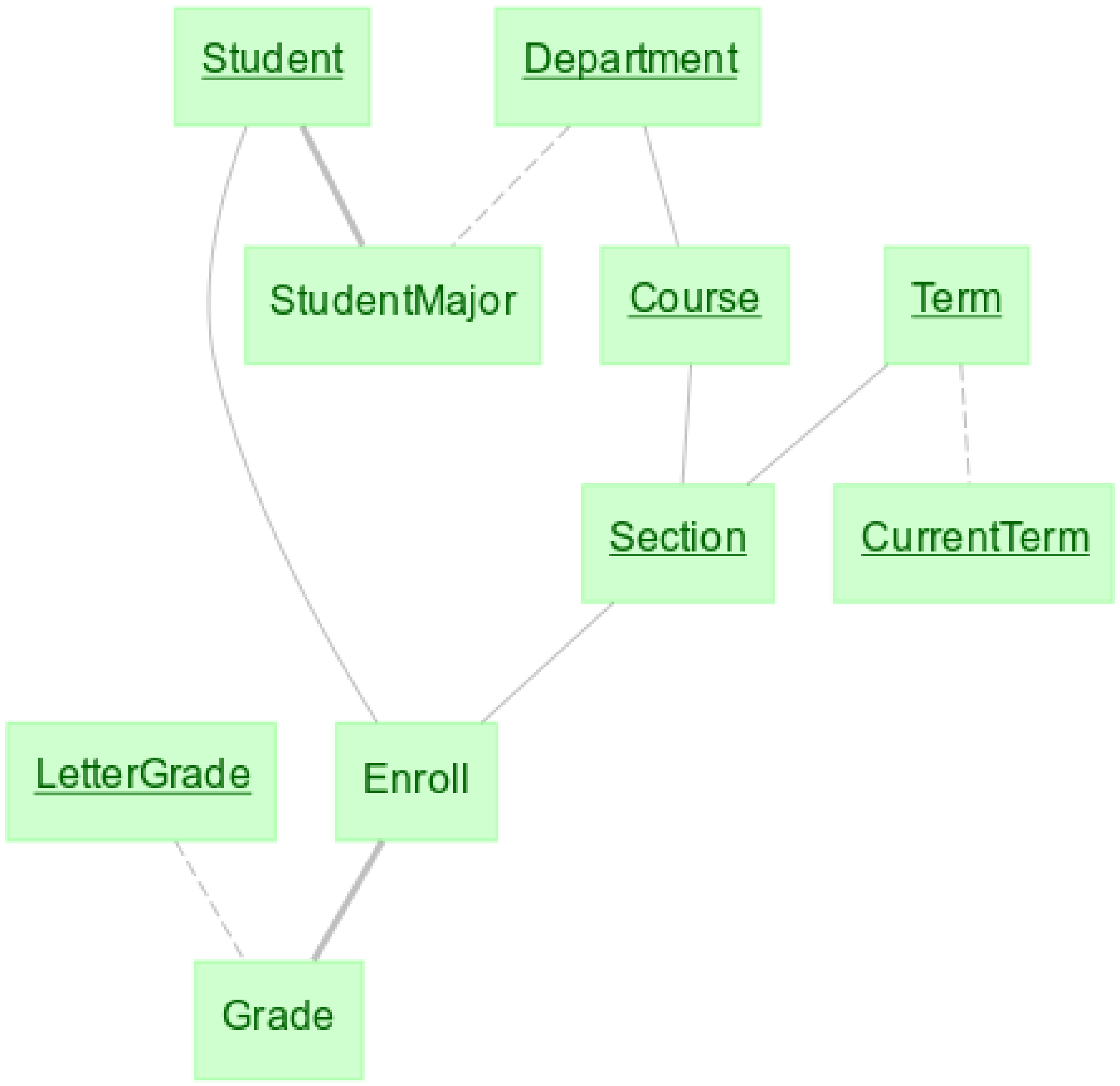
```
# let's plot its diagram
dj.Diagram(schema)
```


`dimitri_university`.`student`
`dimitri_university`.`department`
`dimitri_university`.`student_major`
`dimitri_university`.`course`
`dimitri_university`.`term`
`dimitri_university`.`section`
`dimitri_university`.`current_term`
`dimitri_university`.`letter_grade`
`dimitri_university`.`enroll`
`dimitri_university`.`grade`

You may view the diagram but, at this point, there is no way to interact with these tables. A similar situation arises when another developer has added new tables to the schema but has not yet shared the updated module code with you. Then the diagram will show a mixture of class names and database table names.

Now you may use the `schema.spawn_missing_classes` method to spawn classes into the local namespace for any tables missing their classes:

```
schema.spawn_missing_classes()
dj.Di(schema)
```



Now you may interact with these tables as if they were declared right here in this namespace:

```
# students without majors
Student - StudentMajor
```

student_id university-wide ID number	**first_name**	**last_name**	**sex**	**date_of_birth**	**home_address** mailing street address	**home_city** mailing address	**home_state** US state acronym: e.g. OH	**home_zip** zipcode e.g. 93979-4979	**home_phone** e.g. 414.657.6883x0
1003	Jonathan	Wilson	M	2002-10-19	91101 Summer Park	Port Jacqueline	VT	75616	982.251.1567x2
1005	Richard	Lopez	M	1993-10-16	86210 Brooks Station	West David	IL	96184	+1-385-481-6761x1575
1006	Laura	Hammond	F	1984-12-03	0346 Shannon Motorway	East David	ME	22113	(189)406-2652x

...

Total: 751

6.1.3 Creating a virtual module

Now `spawn_missing_classes` creates the new classes in the local namespace. However, it is often more convenient to import a schema with its Python module, equivalent to the Python command

```
import university as uni
```

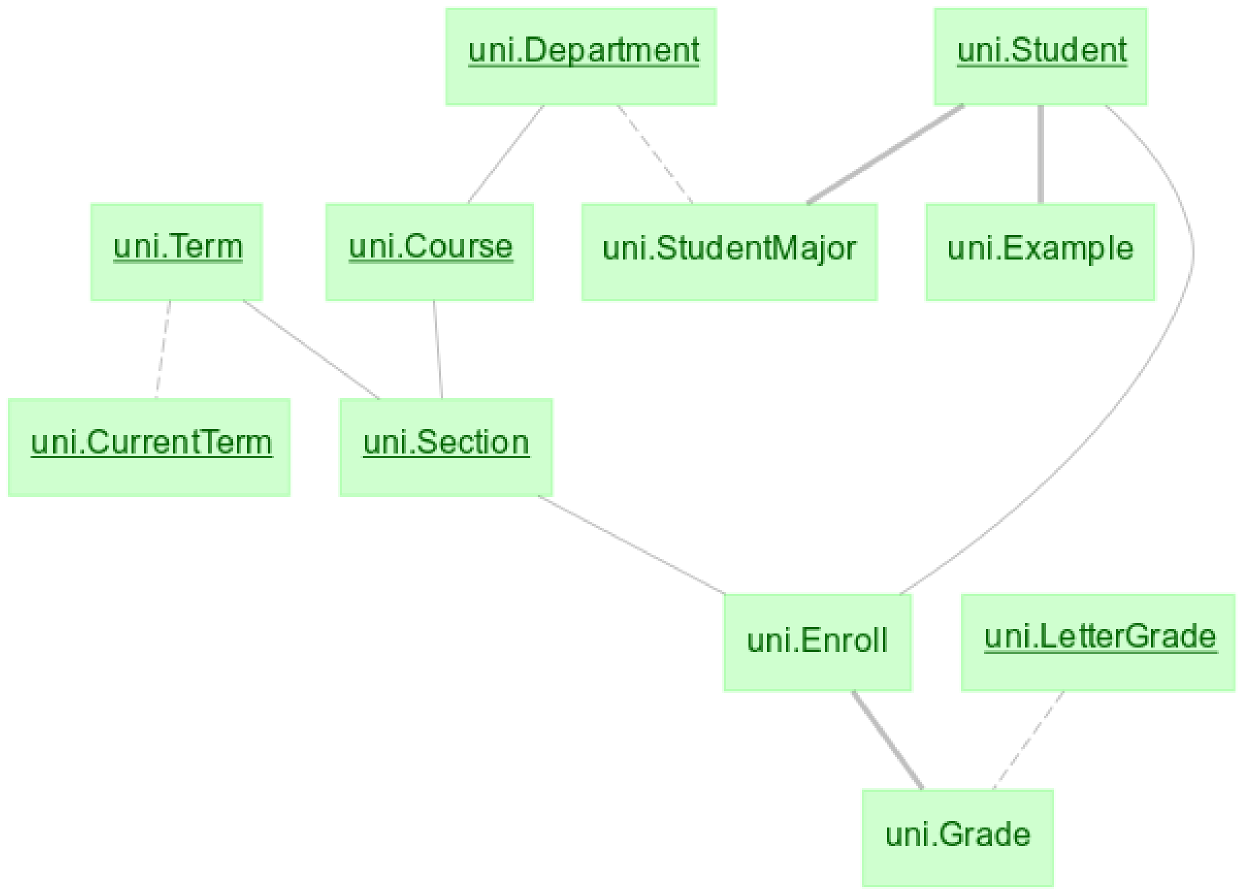
We can mimick this import without having access to `university.py` using the `VirtualModule` class object:

```
import datajoint as dj

uni = dj.VirtualModule('university.py', 'dimitri_university')
```

Connecting dimitri@localhost:3306

Now `uni` behaves as an imported module complete with the schema object and all the table classes.

```
dj.Di(uni)
```



```
uni.Student - uni.StudentMajor
```

student_id university-wide ID number	**first_name**	**last_name**	**sex**	**date_of_birth**	**home_address** mailing street address	**home_city** mailing address	**home_state** US state acronym: e.g. OH	**home_zip** zipcode e.g. 93979-4979	**home_phone** e.g. 414.657.6883x0
1003	Jonathan	Wilson	M	2002-10-19	91101 Summer Park	Port Jacqueline	VT	75616	982.251.1567x2
1005	Richard	Lopez	M	1993-10-16	86210 Brooks Station	West David	IL	96184	+1-385-481-6761x1575
1006	Laura	Hammond	F	1984-12-03	0346 Shannon Motorway	East David	ME	22113	(189)406-2652x

...

Total: 751

`dj.VirtualModule` takes optional arguments.

First, `create_schema=False` assures that an error is raised when the schema does not already exist. Set it to `True` if you want to create an empty schema.

```
dj.VirtualModule('what', 'nonexistent')
```

```
---------------------------------------------------------------------------
DataJointError                            Traceback (most recent call last)
.
.
.
DataJointError: Database named `nonexistent` was not defined. Set argument create_schema=True to␣
↪create it.
```

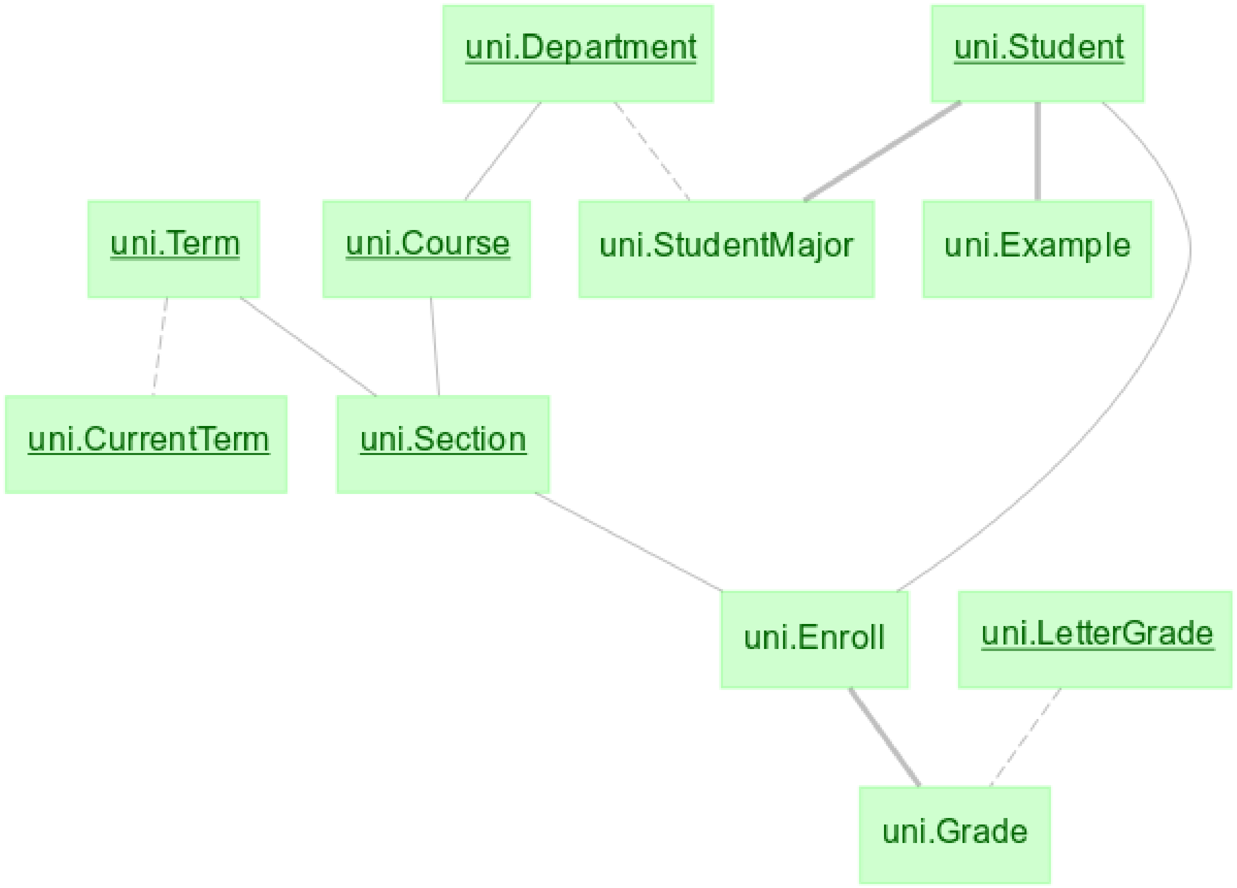
The other optional argument, `create_tables=False` is passed to the schema object. It prevents the use of the schema obect of the virtual module for creating new tables in the existing schema. This is a precautionary measure since virtual modules are often used for completed schemas. You may set this argument to `True` if you wish to add new tables to the existing schema. A more common approach in this scenario would be to create a new schema object and to use the `spawn_missing_classes` function to make the classes available.

However, you if do decide to create new tables in an existing tables using the virtual module, you may do so by using the schema object from the module as the decorator for declaring new tables:

```
uni = dj.VirtualModule('university.py', 'dimitri_university', create_tables=True)
```

```
@uni.schema
class Example(dj.Manual):
    definition = """
    -> uni.Student
    ---
    example : varchar(255)
    """
```

```
dj.Di(uni)
```


uni.Department
uni.Student
uni.Term
uni.Course
uni.StudentMajor
uni.Example
uni.CurrentTerm
uni.Section
uni.Enroll
uni.LetterGrade
uni.Grade

CHAPTER

SEVEN

DATA MANIPULATION

7.1 Manipulation

Data **manipulation** operations change the state of the data stored in the database without modifying the structure of the stored data. These operations include *insert*, *delete*, and *update*.

Data manipulation operations in DataJoint respect the *integrity* constraints.

7.2 Insert

The `insert` method of DataJoint table objects inserts entities into the table.

In Python there is a separate method `insert1` to insert one entity at a time. The entity may have the form of a Python dictionary with key names matching the attribute names in the table.

```
lab.Person.insert1(
       dict(username='alice',
            first_name='Alice',
            last_name='Cooper'))
```

The entity also may take the form of a sequence of values in the same order as the attributes in the table.

```
lab.Person.insert1(['alice', 'Alice', 'Cooper'])
```

Additionally, the entity may be inserted as a NumPy record array or Pandas DataFrame.

The `insert` method accepts a sequence or a generator of multiple entities and is used to insert multiple entities at once.

```
lab.Person.insert([
       ['alice',   'Alice',   'Cooper'],
       ['bob',     'Bob',     'Dylan'],
       ['carol',   'Carol',   'Douglas']])
```

Several optional parameters can be used with `insert`:

`replace` If `True`, replaces the existing entity. (Default `False`.)

`skip_duplicates` If `True`, silently skip duplicate inserts. (Default `False`.)

`ignore_extra_fields` If `False`, fields that are not in the heading raise an error. (Default `False`.)

`allow_direct_insert` If `True`, allows inserts outside of populate calls. Applies only in auto-populated tables. (Default `None`.)

7.2.1 Batched inserts

Inserting a set of entities in a single `insert` differs from inserting the same set of entities one-by-one in a `for` loop in two ways:

1. Network overhead is reduced. Network overhead can be tens of milliseconds per query. Inserting 1000 entities in a single `insert` call may save a few seconds over inserting them individually.
2. The insert is performed as an all-or-nothing transaction. If even one insert fails because it violates any constraint, then none of the entities in the set are inserted.

However, inserting too many entities in a single query may run against buffer size or packet size limits of the database server. Due to these limitations, performing inserts of very large numbers of entities should be broken up into moderately sized batches, such as a few hundred at a time.

7.2.2 Server-side inserts

Data inserted into a table often come from other tables already present on the database server. In such cases, data can be *fetched* from the first table and then inserted into another table, but this results in transfers back and forth between the database and the local system. Instead, data can be inserted from one table into another without transfers between the database and the local system using *queries*.

In the example below, a new schema has been created in preparation for phase two of a project. Experimental protocols from the first phase of the project will be reused in the second phase. Since the entities are already present on the database in the `Protocol` table of the `phase_one` schema, we can perform a server-side insert into `phase_two.Protocol` without fetching a local copy.

```
# Server-side inserts are faster...
phase_two.Protocol.insert(phase_one.Protocol)

# ...than fetching before inserting
protocols = phase_one.Protocol.fetch()
phase_two.Protocol.insert(protocols)
```

7.3 Delete

The `delete` method deletes entities from a table and all dependent entries in dependent tables.

Delete is often used in conjunction with the *restriction* operator to define the subset of entities to delete. Delete is performed as an atomic transaction so that partial deletes never occur.

7.3.1 Examples

```
# delete all entries from tuning.VonMises
tuning.VonMises.delete()

# delete entries from tuning.VonMises for mouse 1010
(tuning.VonMises & 'mouse=1010').delete()

# delete entries from tuning.VonMises except mouse 1010
(tuning.VonMises - 'mouse=1010').delete()
```

7.3.2 Deleting from part tables

Entities in a *part table* are usually removed as a consequence of deleting the master table.

To enforce this workflow, calling `delete` directly on a part table produces an error. In some cases, it may be necessary to override this behavior. To remove entities from a part table without calling `delete` master, use the argument `force=True`.

7.4 Cautious Update

In database programming, the **update** operation refers to modifying the values of individual attributes in an entity within a table without replacing the entire entity. Such an in-place update mechanism is not part of DataJoint's data manipulation model, because it circumvents data *dependency constraints*.

This is not to say that data cannot be changed once they are part of a pipeline. In DataJoint, data are changed by replacing entire entities rather than by updating the values of their attributes. The process of deleting existing entities and inserting new entities with corrected values ensures the *integrity* of the data throughout the pipeline.

This approach applies specifically to automated tables (see *Auto-populate*). However, manual tables are often edited outside DataJoint through other interfaces. It is up to the user's discretion to allow updates in manual tables, and the user must be cognizant of the fact that updates will not trigger re-computation of dependent data.

7.5 Transactions

In some cases, a sequence of several operations must be performed as a single operation: interrupting the sequence of such operations halfway would leave the data in an invalid state. While the sequence is in progress, other processes accessing the database will not see the partial results until the transaction is complete. The sequence make include *data queries* and *manipulations*.

In such cases, the sequence of operations may be enclosed in a **transaction**.

Transactions are formed using the `transaction` property of the connection object. The connection object may be obtained from any table object. The `transaction` property can then be used as a context manager in Python's `with` statement.

For example, the following code inserts matching entries for the master table `Session` and its part table `Session.Experimenter`.

```
# get the connection object
connection = Session.connection

```

(continues on next page)

(continued from previous page)

```python
# insert Session and Session.Experimenter entries in a transaction
with connection.transaction:
    key = {'subject_id': animal_id, 'session_time': session_time}
    Session.insert1({**key, 'brain_region':region, 'cortical_layer':layer})
    Session.Experimenter.insert1({**key, 'experimenter': username})
```

Here, to external observers, both inserts will take effect together upon exiting from the `with` block or will not have any effect at all. For example, if the second insert fails due to an error, the first insert will be rolled back.

CHAPTER

EIGHT

QUERIES

8.1 Query Objects

Data queries retrieve data from the database. A data query is performed with the help of a **query object**, which is a symbolic representation of the query that does not in itself contain any actual data. The simplest query object is an instance of a **table class**, representing the contents of an entire table.

For example, if `experiment.Session` is a DataJoint table class, you can create a query object to retrieve its entire contents as follows:

```
query  = experiment.Session()
```

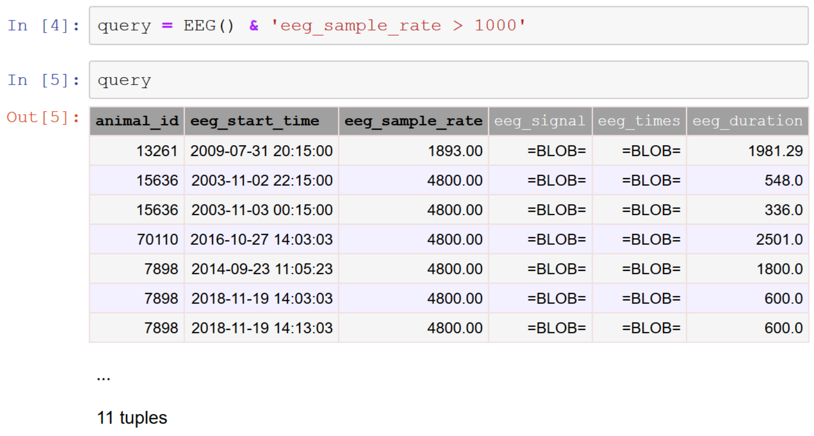
More generally, a query object may be formed as a **query expression** constructed by applying *operators* to other query objects.

For example, the following query retrieves information about all experiments and scans for mouse 102 (excluding experiments with no scans):

```
query = experiment.Session * experiment.Scan & 'animal_id = 102'
```

Note that for brevity, query operators can be applied directly to class objects rather than instance objects so that `experiment.Session` may be used in place of `experiment.Session()`.

You can preview the contents of the query in Python, Jupyter Notebook, or MATLAB by simply displaying the object. In the image below, the object `query` is first defined as a restriction of the table `EEG` by values of the attribute `eeg_sample_rate` greater than 1000 Hz. Displaying the object gives a preview of the entities that will be returned by `query`. Note that this preview only lists a few of the entities that will be returned. Also, the preview does not contain any data for attributes of datatype `blob`.



```
In [4]: query = EEG() & 'eeg_sample_rate > 1000'

In [5]: query

Out[5]:
```

animal_id	eeg_start_time	eeg_sample_rate	eeg_signal	eeg_times	eeg_duration
13261	2009-07-31 20:15:00	1893.00	=BLOB=	=BLOB=	1981.29
15636	2003-11-02 22:15:00	4800.00	=BLOB=	=BLOB=	548.0
15636	2003-11-03 00:15:00	4800.00	=BLOB=	=BLOB=	336.0
70110	2016-10-27 14:03:03	4800.00	=BLOB=	=BLOB=	2501.0
7898	2014-09-23 11:05:23	4800.00	=BLOB=	=BLOB=	1800.0
7898	2018-11-19 14:03:03	4800.00	=BLOB=	=BLOB=	600.0
7898	2018-11-19 14:13:03	4800.00	=BLOB=	=BLOB=	600.0

```
...

11 tuples
```

Fig. 1: Defining a query object and previewing the entities returned by the query.

Once the desired query object is formed, the query can be executed using its *fetch* methods. To **fetch** means to transfer the data represented by the query object from the database server into the workspace of the host language.

s = query.fetch()

Here fetching from the `query` object produces the NumPy record array `s` of the queried data.

8.1.1 Checking for returned entities

The preview of the query object shown above displayed only a few of the entities returned by the query but also displayed the total number of entities that would be returned. It can be useful to know the number of entities returned by a query, or even whether a query will return any entities at all, without having to fetch all the data themselves.

The `bool` function applied to a query object evaluates to `True` if the query returns any entities and to `False` if the query result is empty.

The `len` function applied to a query object determines the number of entities returned by the query.

```
# number of sessions since the start of 2018.
n = len(Session & 'session_date >= "2018-01-01"')
```

8.1.2 Normalization in queries

Query objects adhere to entity *entity normalization* just like the stored tables do. The result of a query is a well-defined entity set with an readily identifiable entity class and designated primary attributes that jointly distinguish any two entities from each other. The query *operators* are designed to keep the result normalized even in complex query expressions.

8.2 Example Schema

The example schema below contains data for a university enrollment system. Information about students, departments, courses, etc. are organized in multiple tables.

Warning: Empty primary keys, such as in the `CurrentTerm` table, are not yet supported by DataJoint. This feature will become available in a future release. See Issue #113 for more information.

```
@schema
class Student (dj.Manual):
  definition = """
  student_id      : int unsigned # university ID
  ---
  first_name      : varchar(40)
  last_name       : varchar(40)
  sex             : enum('F', 'M', 'U')
  date_of_birth   : date
  home_address    : varchar(200) # street address
  home_city       : varchar(30)
  home_state      : char(2) # two-letter abbreviation
  home_zipcode    : char(10)
  home_phone      : varchar(14)
  """

@schema
class Department (dj.Manual):
  definition = """
  dept         : char(6) # abbreviated department name, e.g. BIOL
  ---
  dept_name    : varchar(200) # full department name
  dept_address : varchar(200) # mailing address
  dept_phone   : varchar(14)
  """

@schema
class StudentMajor (dj.Manual):
  definition = """
  -> Student
  ---
  -> Department
  declare_date :  date # when student declared her major
  """

@schema
class Course (dj.Manual):
  definition = """
  -> Department
  course      : int unsigned # course number, e.g. 1010
  ---
  course_name : varchar(200) # e.g. "Cell Biology"
  credits     : decimal(3,1) # number of credits earned by completing the course
  """

@schema
class Term (dj.Manual):
```

(continues on next page)

(continued from previous page)

```
  definition = """
  term_year : year
  term      : enum('Spring', 'Summer', 'Fall')
  """

@schema
class Section (dj.Manual):
  definition = """
  -> Course
  -> Term
  section : char(1)
  ---
  room  :  varchar(12) # building and room code
  """

@schema
class CurrentTerm (dj.Manual):
  definition = """
  ---
  -> Term
  """

@schema
class Enroll (dj.Manual):
  definition = """
  -> Section
  -> Student
  """

@schema
class LetterGrade (dj.Manual):
  definition = """
  grade : char(2)
  ---
  points : decimal(3,2)
  """

@schema
class Grade (dj.Manual):
  definition = """
  -> Enroll
  ---
  -> LetterGrade
  """
```

8.2.1 Example schema ERD

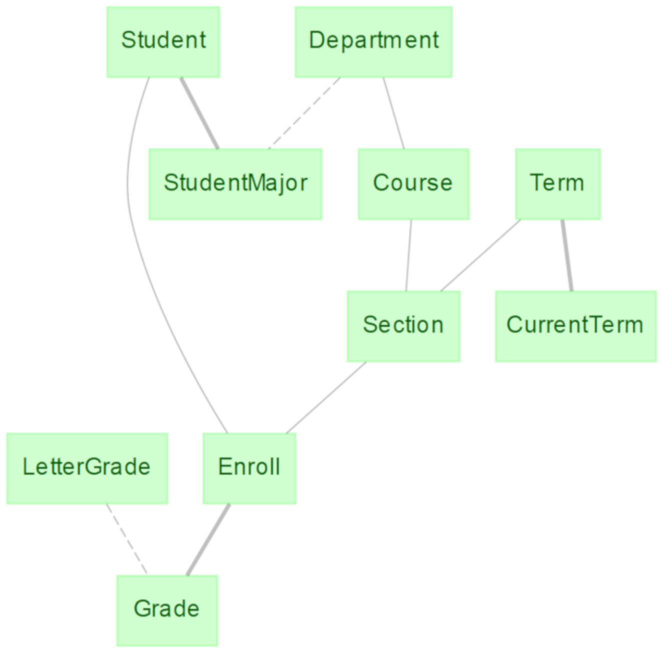


Fig. 2: Example schema for a university database. Tables contain data on students, departments, courses, etc.

8.3 Fetch

Data queries in DataJoint comprise two distinct steps:

1. Construct the `query` object to represent the required data using tables and *operators*.
2. Fetch the data from `query` into the workspace of the host language – described in this section.

Note that entities returned by `fetch` methods are not guaranteed to be sorted in any particular order unless specifically requested. Furthermore, the order is not guaranteed to be the same in any two queries, and the contents of two identical queries may change between two sequential invocations unless they are wrapped in a transaction. Therefore, if you wish to fetch matching pairs of attributes, do so in one `fetch` call.

The examples below are based on the *example schema* for this part of the documentation.

8.3.1 Entire table

The following statement retrieves the entire table as a NumPy recarray.

```
data = query.fetch()
```

To retrieve the data as a list of `dict`:

```
data = query.fetch(as_dict=True)
```

In some cases, the amount of data returned by fetch can be quite large; in these cases it can be useful to use the `size_on_disk` attribute to determine if running a bare fetch would be wise. Please note that it is only currently possible to query the size of entire tables stored directly in the database at this time.

8.3.2 As separate variables

```
name, img = query.fetch1('name', 'image')  # when query has exactly one entity
name, img = query.fetch('name', 'image')  # [name, ...] [image, ...]
```

8.3.3 Primary key values

```
keydict = tab.fetch1("KEY")  # single key dict when tab has exactly one entity
keylist = tab.fetch("KEY")  # list of key dictionaries [{}, ...]
```

`KEY` can also used when returning attribute values as separate variables, such that one of the returned variables contains the entire primary keys.

8.3.4 Usage with Pandas

The `pandas` library is a popular library for data analysis in Python which can easily be used with DataJoint query results. Since the records returned by `fetch()` are contained within a `numpy.recarray`, they can be easily converted to `pandas.DataFrame` objects by passing them into the `pandas.DataFrame` constructor. For example:

```
import pandas as pd
frame = pd.DataFrame(tab.fetch())
```

Calling `fetch()` with the argument `format="frame"` returns results as `pandas.DataFrame` objects with no need for conversion.

```
frame = tab.fetch(format="frame")
```

Returning results as a `DataFrame` is not possible when fetching a particular subset of attributes or when `as_dict` is set to `True`.

8.4 Iteration

The DataJoint model primarily handles data as sets, in the form of tables. However, it can sometimes be useful to access or to perform actions such as visualization upon individual entities sequentially. In DataJoint this is accomplished through iteration.

In the simple example below, iteration is used to display the names and values of the attributes of each entity in the simple table or table expression `tab`.

```
for entity in tab:
    print(entity)
```

This example illustrates the function of the iterator: DataJoint iterates through the whole table expression, returning the entire entity during each step. In this case, each entity will be returned as a `dict` containing all attributes.

At the start of the above loop, DataJoint internally fetches only the primary keys of the entities in `tab`. Since only the primary keys are needed to distinguish between entities, DataJoint can then iterate over the list of primary keys to execute the loop. At each step of the loop, DataJoint uses a single primary key to fetch an entire entity for use in the iteration, such that `print(entity)` will print all attributes of each entity. By first fetching only the primary keys and then fetching each entity individually, DataJoint saves memory at the cost of network overhead. This can be particularly useful for tables containing large amounts of data in secondary attributes.

The memory savings of the above syntax may not be worth the additional network overhead in all cases, such as for tables with little data stored as secondary attributes. In the example below, DataJoint fetches all of the attributes of each entity in a single call and then iterates over the list of entities stored in memory.

```python
for entity in tab.fetch(as_dict=True):
    print(entity)
```

8.5 Operators

Data queries have the form of expressions using operators to derive the desired table. The expressions themselves do not contain any data. They represent the desired data symbolically.

Once a query is formed, the *fetch* methods are used to bring the data into the local workspace. Since the expressions are only symbolic representations, repeated `fetch` calls may yield different results as the state of the database is modified.

DataJoint implements a complete algebra of operators on tables:

operator	notation	meaning
restriction	`A & cond`	The subset of entities from table `A` that meet condition `cond`
restriction	`A - cond`	The subset of entities from table `A` that do not meet condition `cond`
join	`A * B`	Combines all matching information from `A` and `B`
proj	`A.proj(...)`	Selects and renames attributes from `A` or computes new attributes
aggr	`A.aggr(B, ...)`	Same as projection but allows computations based on matching information in `B`
union	`A + B`	All unique entities from both `A` and `B`

8.5.1 Principles of relational algebra

DataJoint's algebra improves upon the classical relational algebra and upon other query languages to simplify and enhance the construction and interpretation of precise and efficient data queries.

1. **Entity integrity**: Data are represented and manipulated in the form of tables representing *well-formed entity sets*. This applies to the inputs and outputs of query operators. The output of a query operator is an entity set with a well-defined entity type, a primary key, unique attribute names, etc.
2. **Algebraic closure**: All operators operate on entity sets and yield entity sets. Thus query expressions may be used as operands in other expressions or may be assigned to variables to be used in other expressions.
3. **Attributes are identified by names**: All attributes have explicit names. This includes results of queries. Operators use attribute names to determine how to perform the operation. The order of the attributes is not significant.

8.5.2 Matching entities

Binary operators in DataJoint are based on the concept of **matching entities**; this phrase will be used throughout the documentation.

> Two entities **match** when they have no common attributes or when their common attributes contain the same values.

Here **common attributes** are those that have the same names in both entities. It is usually assumed that the common attributes are of compatible datatypes to allow equality comparisons.

Another way to phrase the same definition is

> Two entities match when they have no common attributes whose values differ.

It may be conceptually convenient to imagine that all tables always have an additional invisible attribute, `omega` whose domain comprises only one value, 1. Then the definition of matching entities is simplified:

Two entities match when their common attributes contain the same values.

Matching entities can be **merged** into a single entity without any conflicts of attribute names and values.

Examples

This is a matching pair of entities:

user	action
Forrest	run

user	city
Forrest	Mobile, AL

and so is this one:

user	action
Forrest	run

person	city
Jenny	Mobile, AL

but these entities do *not* match:

user	action
Forrest	run

user	city
Jenny	Mobile, AL

8.5.3 Join compatibility

All binary operators with other tables as their two operands require that the operands be **join-compatible**, which means that:

1. All common attributes in both operands (attributes with the same name) must be part of either the primary key or a foreign key.
2. All common attributes in the two relations must be of a compatible datatype for equality comparisons.

These restrictions are introduced both for performance reasons and for conceptual reasons. For performance, they encourage queries that rely on indexes. For conceptual reasons, they encourage database design in which entities in different tables are related to each other by the use of primary keys and foreign keys.

8.6 Restriction

8.6.1 Restriction operators `&` and `-`

The restriction operator `A & cond` selects the subset of entities from `A` that meet the condition `cond`. The exclusion operator `A - cond` selects the complement of restriction, i.e. the subset of entities from `A` that do not meet the condition `cond`.

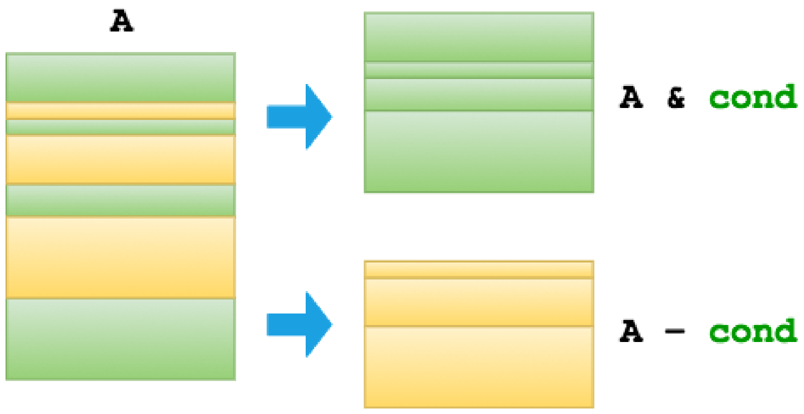


Fig. 3: Restriction and exclusion.

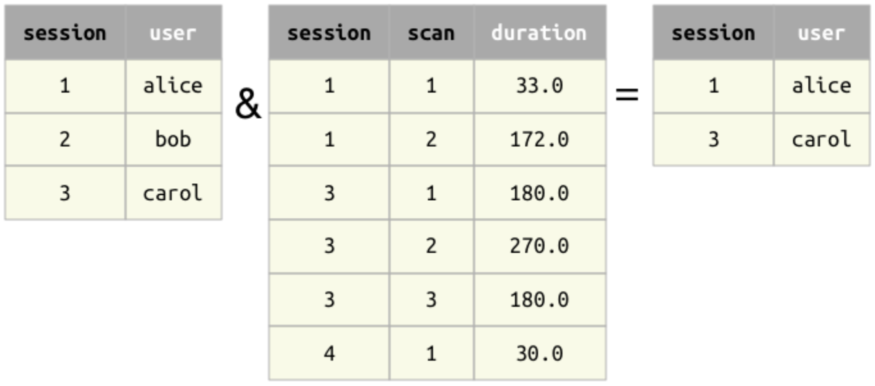
The condition `cond` may be one of the following:

- another table
- a mapping, e.g. `dict`
- an expression in a character string
- a collection of conditions as a `list`, `tuple`, or Pandas `DataFrame`
- a Boolean expression (`True` or `False`)
- an `AndList`
- a `Not` object
- a query expression

As the restriction and exclusion operators are complementary, queries can be constructed using both operators that will return the same results. For example, the queries `A & cond` and `A - Not(cond)` will return the same entities.

8.6.2 Restriction by a table

When restricting table `A` with another table, written `A & B`, the two tables must be **join-compatible** (see *Join compatibility*). The result will contain all entities from `A` for which there exist a matching entity in `B`. Exclusion of table `A` with table `B`, or `A - B`, will contain all entities from `A` for which there are no matching entities in `B`.



session	user
1	alice
2	bob
3	carol

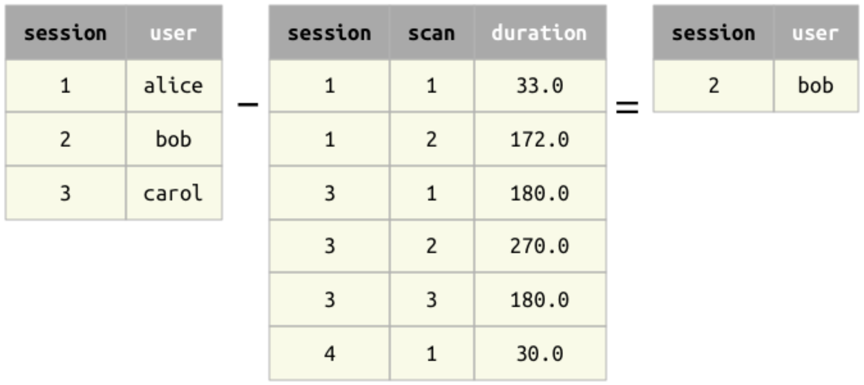
&

session	scan	duration
1	1	33.0
1	2	172.0
3	1	180.0
3	2	270.0
3	3	180.0
4	1	30.0

=

session	user
1	alice
3	carol

Fig. 4: Restriction by another table.



session	user
1	alice
2	bob
3	carol

−

session	scan	duration
1	1	33.0
1	2	172.0
3	1	180.0
3	2	270.0
3	3	180.0
4	1	30.0

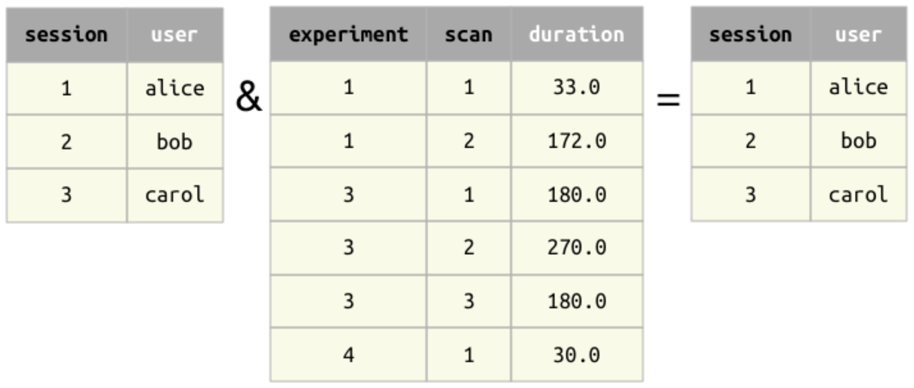
=

session	user
2	bob

Fig. 5: Exclusion by another table.

Restriction by a table with no common attributes

Restriction of table `A` with another table `B` having none of the same attributes as `A` will simply return all entities in `A`, unless `B` is empty as described below. Exclusion of table `A` with `B` having no common attributes will return no entities, unless `B` is empty as described below.



session	user
1	alice
2	bob
3	carol

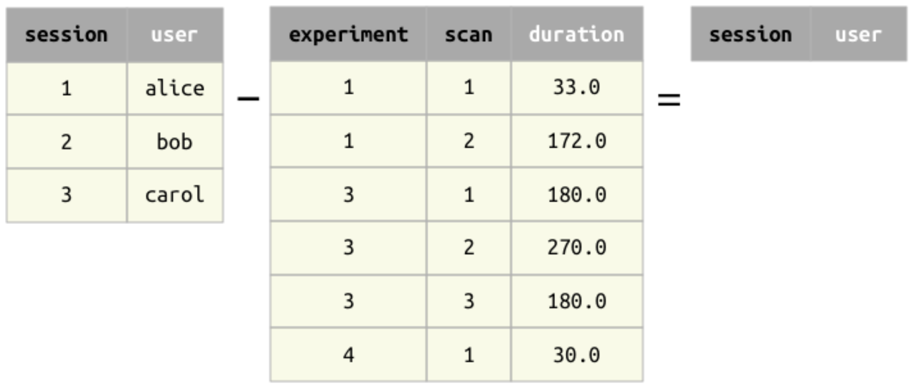
&

experiment	scan	duration
1	1	33.0
1	2	172.0
3	1	180.0
3	2	270.0
3	3	180.0
4	1	30.0

=

session	user
1	alice
2	bob
3	carol

Fig. 6: Restriction by a table having no common attributes.



session	user
1	alice
2	bob
3	carol

−

experiment	scan	duration
1	1	33.0
1	2	172.0
3	1	180.0
3	2	270.0
3	3	180.0
4	1	30.0

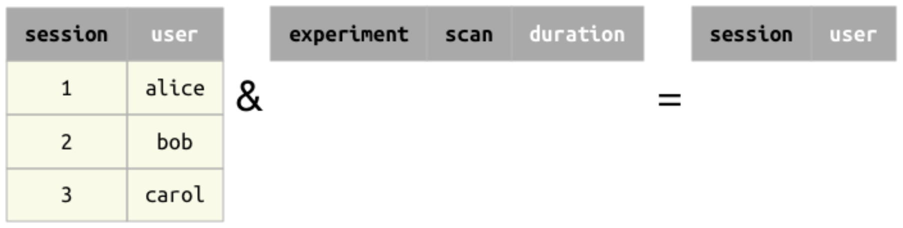
=

session	user

Fig. 7: Exclusion by a table having no common attributes.

Restriction by an empty table

Restriction of table `A` with an empty table will return no entities regardless of whether there are any matching attributes. Exclusion of table `A` with an empty table will return all entities in `A`.



session	user
1	alice
2	bob
3	carol

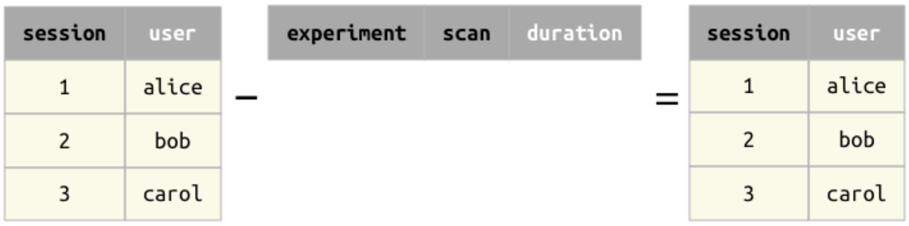
&

experiment	scan	duration

=

session	user

Fig. 8: Restriction by an empty table.



session	user
1	alice
2	bob
3	carol

−

experiment	scan	duration

=

session	user
1	alice
2	bob
3	carol

Fig. 9: Exclusion by an empty table.

8.6.3 Restriction by a mapping

A key-value mapping may be used as an operand in restriction. For each key that is an attribute in `A`, the paired value is treated as part of an equality condition. Any key-value pairs without corresponding attributes in `A` are ignored.

Restriction by an empty mapping or by a mapping with no keys matching the attributes in `A` will return all the entities in `A`. Exclusion by an empty mapping or by a mapping with no matches will return no entities.

For example, let's say that table `Session` has the attribute `session_date` of *datatype* `datetime`. You are interested in sessions from January 1st, 2018, so you write the following restriction query using a mapping.

```
Session & {'session_dat': "2018-01-01"}
```

Our mapping contains a typo omitting the final `e` from `session_date`, so no keys in our mapping will match any attribute in `Session`. As such, our query will return all of the entities of `Session`.

8.6.4 Restriction by a string

Restriction can be performed when `cond` is an explicit condition on attribute values, expressed as a string. Such conditions may include arithmetic operations, functions, range tests, etc. Restriction of table `A` by a string containing an attribute not found in table `A` produces an error.

```
# All the sessions performed by Alice
Session & 'user = "Alice"'

# All the experiments at least one minute long
Experiment & 'duration >= 60'
```

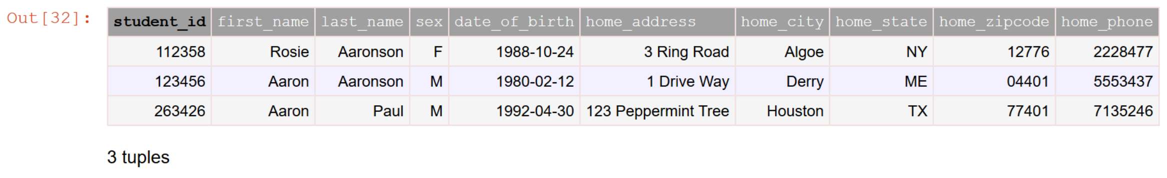
8.6.5 Restriction by a collection

A collection can be a list, a tuple, or a Pandas `DataFrame`.

```
# a list:
cond_list = ['first_name = "Aaron"', 'last_name = "Aaronson"']

# a tuple:
cond_tuple = ('first_name = "Aaron"', 'last_name = "Aaronson"')

# a dataframe:
import pandas as pd
cond_frame = pd.DataFrame(
            data={'first_name': ['Aaron'], 'last_name': ['Aaronson']})
```

When `cond` is a collection of conditions, the conditions are applied by logical disjunction (logical OR). Thus, restriction of table `A` by a collection will return all entities in `A` that meet *any* of the conditions in the collection. For example, if you restrict the `Student` table by a collection containing two conditions, one for a first and one for a last name, your query will return any students with a matching first name *or* a matching last name.

```
Student() & ['first_name = "Aaron"', 'last_name = "Aaronson"']
```



Out[32]:

student_id	first_name	last_name	sex	date_of_birth	home_address	home_city	home_state	home_zipcode	home_phone
112358	Rosie	Aaronson	F	1988-10-24	3 Ring Road	Algoe	NY	12776	2228477
123456	Aaron	Aaronson	M	1980-02-12	1 Drive Way	Derry	ME	04401	5553437
263426	Aaron	Paul	M	1992-04-30	123 Peppermint Tree	Houston	TX	77401	7135246

3 tuples

Fig. 10: Restriction by a collection, returning all entities matching any condition in the collection.

Restriction by an empty collection returns no entities. Exclusion of table `A` by an empty collection returns all the entities of `A`.

8.6.6 Restriction by a Boolean expression

`A & True` and `A - False` are equivalent to `A`.

`A & False` and `A - True` are empty.

8.6.7 Restriction by an `AndList`

The special function `dj.AndList` represents logical conjunction (logical AND). Restriction of table `A` by an `AndList` will return all entities in `A` that meet *all* of the conditions in the list. `A & dj.AndList([c1, c2, c3])` is equivalent to `A & c1 & c2 & c3`. Usually, it is more convenient to simply write out all of the conditions, as `A & c1 & c2 & c3`. However, when a list of conditions has already been generated, the list can simply be passed as the argument to `dj.AndList`.

Restriction of table `A` by an empty `AndList`, as in `A & dj.AndList([])`, will return all of the entities in `A`. Exclusion by an empty `AndList` will return no entities.

8.6.8 Restriction by a `Not` object

The special function `dj.Not` represents logical negation, such that `A & dj.Not(cond)` is equivalent to `A - cond`.

8.6.9 Restriction by a query

Restriction by a query object is a generalization of restriction by a table (which is also a query object), because DataJoint queries always produce well-defined entity sets, as described in *entity normalization*. As such, restriction by queries follows the same behavior as restriction by tables described above.

The example below creates a query object corresponding to all the sessions performed by the user Alice. The `Experiment` table is then restricted by the query object, returning all the experiments that are part of sessions performed by Alice.

```
query = Session & 'user = "Alice"'
Experiment & query
```

8.7 Join

8.7.1 Join operator `*`

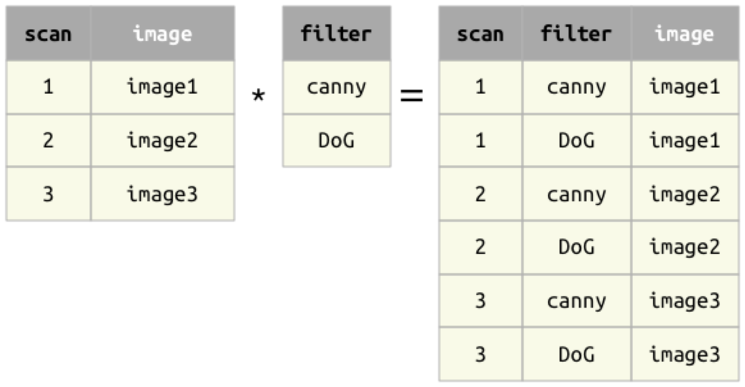
The Join operator `A * B` combines the matching information in `A` and `B`. The result contains all matching combinations of entities from both arguments.

Principles of joins

1. The operands `A` and `B` must be **join-compatible**.
2. The primary key of the result is the union of the primary keys of the operands.

Examples of joins

Example 1 : When the operands have no common attributes, the result is the cross product – all combinations of entities.



scan	image
1	image1
2	image2
3	image3

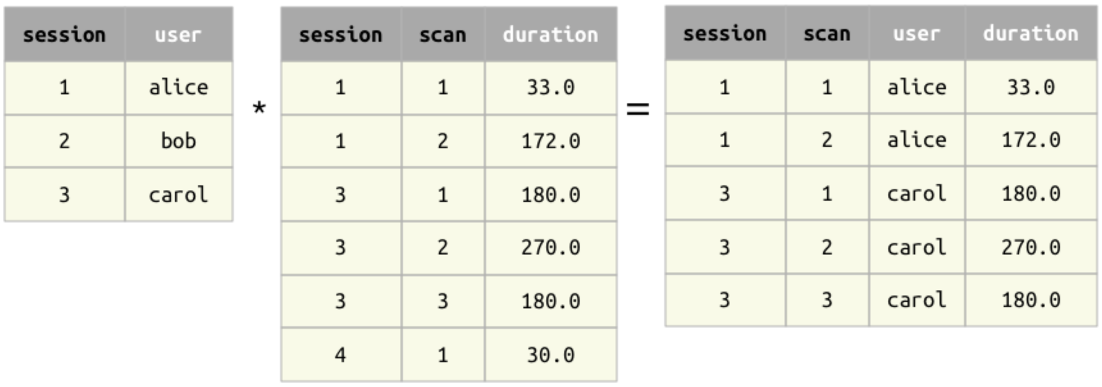
*

filter
canny
DoG

=

scan	filter	image
1	canny	image1
1	DoG	image1
2	canny	image2
2	DoG	image2
3	canny	image3
3	DoG	image3

Example 2 : When the operands have common attributes, only entities with matching values are kept.



session	user
1	alice
2	bob
3	carol

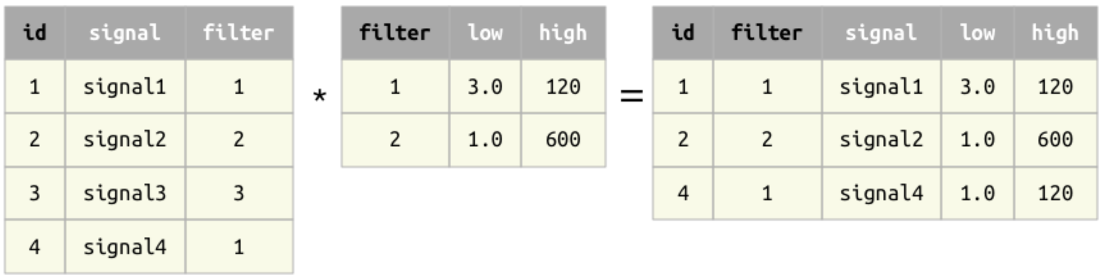
*

session	scan	duration
1	1	33.0
1	2	172.0
3	1	180.0
3	2	270.0
3	3	180.0
4	1	30.0

=

session	scan	user	duration
1	1	alice	33.0
1	2	alice	172.0
3	1	carol	180.0
3	2	carol	270.0
3	3	carol	180.0

Example 3 : Joining on secondary attribute.



id	signal	filter
1	signal1	1
2	signal2	2
3	signal3	3
4	signal4	1

*

filter	low	high
1	3.0	120
2	1.0	600

=

id	filter	signal	low	high
1	1	signal1	3.0	120
2	2	signal2	1.0	600
4	1	signal4	1.0	120

Properties of join

1. When `A` and `B` have the same attributes, the join `A * B` becomes equivalent to the set intersection `A` ∩ `B`. Hence, DataJoint does not need a separate intersection operator.
2. Commutativity: `A * B` is equivalent to `B * A`.
3. Associativity: `(A * B) * C` is equivalent to `A * (B * C)`.

8.8 Proj

The `proj` operator represents **projection** and is used to select attributes (columns) from a table, to rename them, or to create new calculated attributes.

8.8.1 Simple projection

The simple projection selects a subset of attributes of the original table. However, the primary key attributes are always included.

Using the *example schema*, let table `department` have attributes **dept**, *dept_name*, *dept_address*, and *dept_phone*. The primary key attribute is in bold.

Then `department.proj()` will have attribute **dept**.

`department.proj('dept')` will have attribute **dept**.

`department.proj('dept_name', 'dept_phone')` will have attributes **dept**, *dept_name*, and *dept_phone*.

8.8.2 Renaming

In addition to selecting attributes, `proj` can rename them. Any attribute can be renamed, including primary key attributes.

This is done using keyword arguments: `tab.proj(new_attr='old_attr')`

For example, let table `tab` have attributes **mouse**, **session**, *session_date*, *stimulus*, and *behavior*. The primary key attributes are in bold.

Then

```
tab.proj(animal='mouse', 'stimulus')
```

will have attributes **animal**, **session**, and *stimulus*.

Renaming is often used to control the outcome of a *join*. For example, let `tab` have attributes **slice**, and **cell**. Then `tab * tab` will simply yield `tab`. However,

```
tab * tab.proj(other='cell')
```

yields all ordered pairs of all cells in each slice.

8.8.3 Calculations

In addition to selecting or renaming attributes, `proj` can compute new attributes from existing ones.

For example, let `tab` have attributes `mouse`, `scan`, `surface_z`, and `scan_z`. To obtain the new attribute `depth` computed as `scan_z - surface_z` and then to restrict to `depth > 500`:

```
tab.proj(depth='scan_z-surface_z') & 'depth > 500'
```

Calculations are passed to SQL and are not parsed by DataJoint. For available functions, you may refer to the MySQL documentation.

8.9 Aggr

Aggregation, performed with the `aggr` operator, is a special form of `proj` with the additional feature of allowing aggregation calculations on another table. It has the form `tab.aggr(other, ...)` where `other` is another table. Without the argument `other`, `aggr` and `proj` are exactly equivalent. Aggregation allows adding calculated attributes to each entity in `tab` based on aggregation functions over attributes in the *matching* entities of `other`.

Aggregation functions include `count`, `sum`, `min`, `max`, `avg`, `median`, `percentile`, `stdev`, `var`, and others. Aggregation functions can only be used in the definitions of new attributes within the `aggr` operator.

As with `proj`, the output of `aggr` has the same entity class, the same primary key, and the same number of elements as `tab`. Primary key attributes are always included in the output and may be renamed, just like in `proj`.

8.9.1 Examples

```
# Number of students in each course section
Section.aggr(Enroll, n="count(*)")
# Average grade in each course
Course.aggr(Grade * LetterGrade, avg_grade="avg(points)")
```

8.10 Union

The union operator is not yet implemented – this page serves as the specification for the upcoming implementation. Union is rarely needed in practice.

8.10.1 Union operator `+`

The result of the union operator `A + B` contains all the entities from both operands. *Entity normalization* requires that the operands in a union both belong to the same entity type with the same primary key using homologous attributes. In the absence of any secondary attributes, the result of a union is the simple set union.

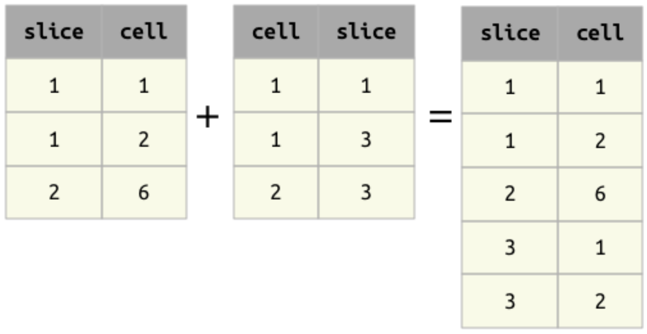
When secondary attributes are present, they must have the same names and datatypes in both operands. The two operands must also be **disjoint**, without any duplicate primary key values across both inputs. These requirements prevent ambiguity of attribute values and preserve entity identity.

Principles of union

1. As in all operators, the order of the attributes in the operands is not significant.
2. Operands `A` and `B` must have the same primary key attributes. Otherwise, an error will be raised.
3. Operands `A` and `B` may not have any common non-key attributes. Otherwise, an error will be raised.
4. The result `A + B` will have the same primary key as `A` and `B`.
5. The result `A + B` will have all the non-key attributes from both `A` and `B`.
6. For entities that are found in both `A` and `B` (based on the primary key), the secondary attributes will be filled from the corresponding entities in `A` and `B`.
7. For entities that are only found in either `A` or `B`, the other operand's secondary attributes will filled with null values.

Examples of union

Example 1 : Note that the order of the attributes does not matter.



slice	cell
1	1
1	2
2	6

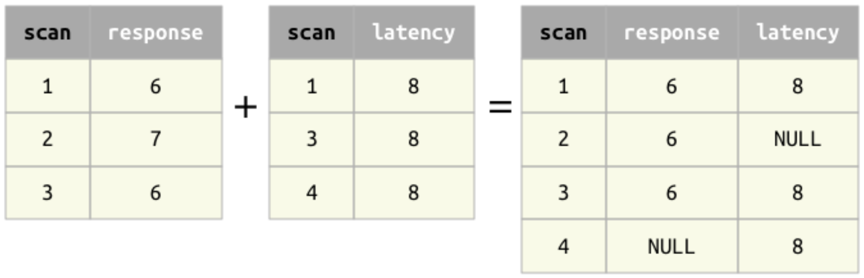
+

cell	slice
1	1
1	3
2	3

=

slice	cell
1	1
1	2
2	6
3	1
3	2

Example 2 : Non-key attributes are combined from both tables and filled with NULLs when missing.



scan	response
1	6
2	7
3	6

+

scan	latency
1	8
3	8
4	8

=

scan	response	latency
1	6	8
2	6	NULL
3	6	8
4	NULL	8

Properties of union

1. Commutative: `A + B` is equivalent to `B + A`.
2. Associative: `(A + B) + C` is equivalent to `A + (B + C)`.

8.11 Universal Sets

All *query operators* are designed to preserve the entity types of their inputs. However, some queries require creating a new entity type that is not represented by any stored tables. This means that a new entity type must be explicitly defined as part of the query. Universal sets fulfill this role.

Universal sets are used in DataJoint to define virtual tables with arbitrary primary key structures for use in query expressions. A universal set, defined using class `dj.U`, denotes the set of all possible entities with given attributes of any possible datatype. Universal sets allow query expressions using virtual tables when no suitable base table exists. Attributes of universal sets are allowed to be matched to any namesake attributes, even those that do not come from the same initial source.

For example, you may like to query the university database for the complete list of students' home cities, along with the number of students from each city. The *schema* for the university database does not have a table for cities and states. A virtual table can fill the role of the nonexistent base table, allowing queries that would not be possible otherwise.

```python
# All home cities of students
dj.U('home_city', 'home_state') & Student

# Total number of students from each city
dj.U('home_city', 'home_state').aggr(Student, n="count(*)")

# Total number of students from each state
```

(continues on next page)

(continued from previous page)

```
U('home_state').aggr(Student, n="count(*)")

# Total number of students in the database
U().aggr(Student, n="count(*)")
```

The result of aggregation on a universal set is restricted to the entities with matches in the aggregated table, such as `Student` in the example above. In other words, `X.aggr(A, ...)` is interpreted as `(X & A).aggr(A, ...)` for universal set `X`. All attributes of a universal set are considered primary.

Universal sets should be used sparingly when no suitable base tables already exist. In some cases, defining a new base table can make queries clearer and more semantically constrained.

CHAPTER

NINE

COMPUTATION

9.1 Auto-populate

Auto-populated tables are used to define, execute, and coordinate computations in a DataJoint pipeline.

Tables in the initial portions of the pipeline are populated from outside the pipeline. In subsequent steps, computations are performed automatically by the DataJoint pipeline in auto-populated tables.

Computed tables belong to one of the two auto-populated *data tiers*: `dj.Imported` and `dj.Computed`. DataJoint does not enforce the distinction between imported and computed tables: the difference is purely semantic, a convention for developers to follow. If populating a table requires access to external files such as raw storage that is not part of the database, the table is designated as **imported**. Otherwise it is **computed**.

Auto-populated tables are defined and queried exactly as other tables. (See *Manual Tables*.) Their data definition follows the same *definition syntax*.

9.1.1 Make

For auto-populated tables, data should never be entered using *insert* directly. Instead these tables must define the callback method `make(self, key)`. The `insert` method then can only be called on `self` inside this callback method.

Imagine that there is a table `test.Image` that contains 2D grayscale images in its `image` attribute. Let us define the computed table, `test.FilteredImage` that filters the image in some way and saves the result in its `filtered_image` attribute.

The class will be defined as follows.

```
@schema
class FilteredImage(dj.Computed):
    definition = """
    # Filtered image
    -> Image
    ---
    filtered_image : longblob
    """

    def make(self, key):
        img = (test.Image & key).fetch1['image']
        key['filtered_image'] = myfilter(img)
        self.insert(key)
```

The `make` method receives one argument: the dict `key` containing the primary key value of an element of *key source* to be worked on.

The `make` method received one argument: the `key` of type `struct` in MATLAB and `dict` in Python. The key represents the partially filled entity, usually already containing the *primary key* attributes of the key source.

The `make` callback does three things:

1. *Fetches* data from tables upstream in the pipeline using the `key` for *restriction*.
2. Computes and adds any missing attributes to the fields already in `key`.
3. Inserts the entire entity into `self`.

`make` may populate multiple entities in one call when `key` does not specify the entire primary key of the populated table.

9.1.2 Populate

The inherited `populate` method of `dj.Imported` and `dj.Computed` automatically calls `make` for every key for which the auto-populated table is missing data.

The `FilteredImage` table can be populated as

```
FilteredImage.populate()
```

The progress of long-running calls to `populate()` in datajoint-python can be visualized by adding the `display_progress=True` argument to the populate call.

Note that it is not necessary to specify which data needs to be computed. DataJoint will call `make`, one-by-one, for every key in `Image` for which `FilteredImage` has not yet been computed.

Chains of auto-populated tables form computational pipelines in DataJoint.

9.1.3 Populate options

The `populate` method accepts a number of optional arguments that provide more features and allow greater control over the method's behavior.

- `restrictions` - A list of restrictions, restricting as `(tab.key_source & AndList(restrictions)) - tab.proj()`. Here `target` is the table to be populated, usually `tab` itself.
- `suppress_errors` - If `True`, encountering an error will cancel the current `make` call, log the error, and continue to the next `make` call. Error messages will be logged in the job reservation table (if `reserve_jobs` is `True`) and returned as a list. See also `return_exception_objects` and `reserve_jobs`. Defaults to `False`.
- `return_exception_objects` - If `True`, error objects are returned instead of error messages. This applies only when `suppress_errors` is `True`. Defaults to `False`.
- `reserve_jobs` - If `True`, reserves job to indicate to other distributed processes. The job reservation table may be access as `schema.jobs`. Errors are logged in the jobs table. Defaults to `False`.
- `order` - The order of execution, either `"original"`, `"reverse"`, or `"random"`. Defaults to `"original"`.

- `display_progress` - If `True`, displays a progress bar. Defaults to `False`.
- `limit` - If not `None`, checks at most this number of keys. Defaults to `None`.
- `max_calls` - If not `None`, populates at most this many keys. Defaults to `None`, which means no limit.

9.1.4 Progress

The method `table.progress` reports how many `key_source` entries have been populated and how many remain. Two optional parameters allow more advanced use of the method. A parameter of restriction conditions can be provided, specifying which entities to consider. A Boolean parameter `display` (default is `True`) allows disabling the output, such that the numbers of remaining and total entities are returned but not printed.

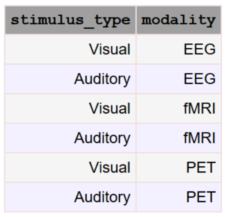
9.2 Key Source

9.2.1 Default key source

Key source refers to the set of primary key values over which *autopopulate* iterates, calling the `make` method at each iteration. Each `key` from the key source is passed to the table's `make` call. By default, the key source for a table is the *join* of its primary *dependencies*.

For example, consider a schema with three tables. The `Stimulus` table contains one attribute `stimulus_type` with one of two values, "Visual" or "Auditory". The `Modality` table contains one attribute `modality` with one of three values, "EEG", "fMRI", and "PET". The `Protocol` table has primary dependencies on both the `Stimulus` and `Modality` tables.

The key source for `Protocol` will then be all six combinations of `stimulus_type` and `modality` as shown in the figure below.



stimulus_type	modality
Visual	EEG
Auditory	EEG
Visual	fMRI
Auditory	fMRI
Visual	PET
Auditory	PET

9.2.2 Custom key source

A custom key source can be configured by setting the `key_source` property within a table class, after the `definition` string.

Any *query object* can be used as the key source. In most cases the new key source will be some alteration of the default key source. Custom key sources often involve restriction to limit the key source to only relevant entities. Other designs may involve using only one of a table's primary dependencies.

In the example below, the `EEG` table depends on the `Recording` table that lists all recording sessions. However, the `populate` method of `EEG` should only ingest recordings where the `recording_type` is `EEG`. Setting a custom key source prevents the `populate` call from iterating over recordings of the wrong type.

A custom key source can be configured by setting the `key_source` property within a table class, after the `definition` string.

9.3 Master-Part Relationship

Often an entity in one table is inseparably associated with a group of entities in another, forming a **master-part** relationship. The master-part relationship ensures that all parts of a complex representation appear together or not at all. This has become one of the most powerful data integrity principles in DataJoint.

As an example, imagine segmenting an image to identify regions of interest. The resulting segmentation is inseparable from the ROIs that it produces. In this case, the two tables might be called `Segmentation` and `Segmentation.ROI`.

In Python, the master-part relationship is expressed by making the part a nested class of the master. The part is subclassed from `dj.Part` and does not need the `@schema` decorator.

```python
@schema
class Segmentation(dj.Computed):
    definition = """  # image segmentation
    -> Image
    """

    class ROI(dj.Part):
        definition = """  # Region of interest resulting from segmentation
        -> Segmentation
        roi  : smallint   # roi number
        ---
        roi_pixels  : longblob   #  indices of pixels
        roi_weights : longblob   #  weights of pixels
        """

    def make(self, key):
        image = (Image & key).fetch1['image']
        self.insert1(key)
        count = itertools.count()
        Segmentation.ROI.insert(
                dict(key, roi=next(count), roi_pixel=roi_pixels, roi_weights=roi_weights)
                for roi_pixels, roi_weights in mylib.segment(image))
```

9.3.1 Populating

Master-part relationships can form in any data tier, but DataJoint observes them more strictly for auto-populated tables. To populate both the master `Segmentation` and the part `Segmentation.ROI`, it is sufficient to call the `populate` method of the master:

```python
Segmentation.populate()
```

Note that the entities in the master and the matching entities in the part are inserted within a single `make` call of the master, which means that they are a processed inside a single transactions: either all are inserted and committed or the entire transaction is rolled back. This ensures that partial results never appear in the database.

For example, imagine that a segmentation is performed, but an error occurs halfway through inserting the results. If this situation were allowed to persist, then it might appear that 20 ROIs were detected where 45 had actually been found.

9.3.2 Deleting

To delete from a master-part pair, one should never delete from the part tables directly. The only valid method to delete from a part table is to delete the master. This has been an unenforced rule, but upcoming versions of DataJoint will prohibit direct deletes from the master table. DataJoint's *delete* operation is also enclosed in a transaction.

Together, the rules of master-part relationships ensure a key aspect of data integrity: results of computations involving multiple components and steps appear in their entirety or not at all.

9.3.3 Multiple parts

The master-part relationship cannot be chained or nested. DataJoint does not allow part tables of other part tables per se. However, it is common to have a master table with multiple part tables that depend on each other. For example:

```
@schema
class ArrayResponse(dj.Computed):
definition = """
array: int
"""

  class ElectrodeResponse(dj.Part):
  definition = """
  -> master
  electrode: int    # electrode number on the probe
  """

  class ChannelResponse(dj.Part):
  definition = """
  -> ElectrodeResponse
  channel: int
  ---
  response: longblob  # response of a channel
  """
```

Conceptually, one or more channels belongs to an electrode, and one or more electrodes belong to an array. This example assumes that information about an array's response (which consists ultimately of the responses of multiple electrodes each consisting of multiple channel responses) including it's electrodes and channels are entered together.

9.4 Transactions in Make

Each call of the *make* method is enclosed in a transaction. DataJoint users do not need to explicitly manage transactions but must be aware of their use.

Transactions produce two effects:

First, the state of the database appears stable within the `make` call throughout the transaction: two executions of the same query will yield identical results within the same `make` call.

Second, any changes to the database (inserts) produced by the `make` method will not become visible to other processes until the `make` call completes execution. If the `make` method raises an exception, all changes made so far will be discarded and will never become visible to other processes.

Transactions are particularly important in maintaining *group integrity* with *master-part relationships*. The `make` call of a master table first inserts the master entity and then inserts all the matching part entities in the part tables. None of the entities become visible to other processes until the entire `make` call completes, at which point they all become visible.

9.5 Distributed Computing

9.5.1 Job reservations

Running `populate` on the same table on multiple computers will causes them to attempt to compute the same data all at once. This will not corrupt the data since DataJoint will reject any duplication. One solution could be to cause the different computing nodes to populate the tables in random order. This would reduce some collisions but not completely prevent them.

To allow efficient distributed computing, DataJoint provides a built-in job reservation process. When `dj.Computed` tables are auto-populated using job reservation, a record of each ongoing computation is kept in a schema-wide `jobs` table, which is used internally by DataJoint to coordinate the auto-population effort among multiple computing processes.

Job reservations are activated by setting the keyword argument `reserve_jobs=True` in `populate` calls.

With job management enabled, the `make` method of each table class will also consult the `jobs` table for reserved jobs as part of determining the next record to compute and will create an entry in the `jobs` table as part of the attempt to compute the resulting record for that key. If the operation is a success, the record is removed. In the event of failure, the job reservation entry is updated to indicate the details of failure. Using this simple mechanism, multiple processes can participate in the auto-population effort without duplicating computational effort, and any errors encountered during the course of the computation can be individually inspected to determine the cause of the issue.

As part of DataJoint, the jobs table can be queried using native DataJoint syntax. For example, to list the jobs currently being run:

```
In [1]: schema.jobs
Out[1]:
*table_name    *key_hash      status       error_message  user           host           pid     ␣
↪connection_id  timestamp      key        error_stack
+------------+ +------------+ +----------+ +------------+ +------------+ +------------+ +--------+␣
↪+------------+ +------------+ +--------+ +------------+
__job_results  e4da3b7fbbce23 reserved                    datajoint@localhos localhost     15571  ␣
↪  59             2017-09-04 14: <BLOB>     <BLOB>
 (2 tuples)
```

The above output shows that a record for the `JobResults` table is currently reserved for computation, along with various related details of the reservation, such as the MySQL connection ID, client user and host, process ID on the remote system, timestamp, and the key for the record that the job is using for its computation. Since DataJoint table keys can be of varying types, the key is stored in a binary format to allow the table to store arbitrary types of record key data. The subsequent sections will discuss querying the jobs table for key data.

As mentioned above, jobs encountering errors during computation will leave their record reservations in place, and update the reservation record with details of the error.

For example, if a Python process is interrupted via the keyboard, a KeyboardError will be logged to the database as follows:

```
In [2]: schema.jobs
Out[2]:
*table_name    *key_hash      status     error_message  user           host           pid     ␣
↪connection_id  timestamp      key        error_stack
```

(continues on next page)

(continued from previous page)

```
+------------+ +------------+ +--------+ +------------+ +------------+ +------------+ +--------+ +--
↪----------+ +------------+ +--------+ +------------+
__job_results  3416a75f4cea91 error      KeyboardInterr datajoint@localhos localhost     15571    ␣
↪59             2017-09-04 14: <BLOB>     <BLOB>
 (1 tuples)
```

By leaving the job reservation record in place, the error can be inspected, and if necessary the corresponding `dj.Computed` update logic can be corrected. From there the jobs entry can be cleared, and the computation can then be resumed. In the meantime, the presence of the job reservation will prevent this particular record from being processed during subsequent auto-population calls. Inspecting the job record for failure details can proceed much like any other DataJoint query.

For example, given the above table, errors can be inspected as follows:

```
In [3]: (schema.jobs & 'status="error"' ).fetch(as_dict=True)
Out[3]:
[OrderedDict([('table_name', '__job_results'),
              ('key_hash', 'c81e728d9d4c2f636f067f89cc14862c'),
              ('status', 'error'),
              ('key', rec.array([(2,)],
                        dtype=[('id', 'O')])),
              ('error_message', 'KeyboardInterrupt'),
              ('error_stack', None),
              ('user', 'datajoint@localhost'),
              ('host', 'localhost'),
              ('pid', 15571),
              ('connection_id', 59),
              ('timestamp', datetime.datetime(2017, 9, 4, 15, 3, 53))])]
```

This particular error occurred when processing the record with ID `2`, resulted from a *KeyboardInterrupt*, and has no additional error trace.

After any system or code errors have been resolved, the table can simply be cleaned of errors and the computation rerun.

For example:

```
In [4]: (schema.jobs & 'status="error"' ).delete()
```

In some cases, it may be preferable to inspect the jobs table records using populate keys. Since job keys are hashed and stored as a blob in the jobs table to support the varying types of keys, we need to query using the key hash instead of simply using the raw key data.

9.5.2 Managing connections

The DataJoint method `dj.kill` allows for viewing and termination of database connections. Restrictive conditions can be used to identify specific connections. Restrictions are specified as strings and can involve any of the attributes of `information_schema.processlist`: `ID`, `USER`, `HOST`, `DB`, `COMMAND`, `TIME`, `STATE`, and `INFO`.

Examples:

`dj.kill('HOST LIKE "%compute%"')` lists only connections from hosts containing “compute”.

`dj.kill('TIME > 600')` lists only connections older than 10 minutes.

A list of connections meeting the restriction conditions (if present) are presented to the user, along with the option to kill processes.

For example, to sort the output by hostname in descending order:

```
In [3]: dj.kill(None, None, 'host desc')
Out[3]:
  ID USER         HOST          STATE         TIME    INFO
+--+ +----------+ +-----------+ +-----------+ +-----+
  33 chris        localhost:54840                1261  None
  17 chris        localhost:54587                3246  None
   4 event_scheduler localhost    Waiting on empty queue  187180  None
process to kill or "q" to quit > q
```