**SchwartzLab Data Analysis in Datajoint for MATLAB**

The basics.

Datajoint is a data management system for neuroscience that originated in the lab of Andreas Tolias at Baylor. It is built on the framework of an SQL relational database – an extremely broadly used and scalable structure that supports large companies and organizations around the world. Understanding the core concepts of relational databases and some basic datajoint commands will certainly serve you well in the lab (and beyond), but this documentation focuses mainly on the SchwartzLab implementation that I have built on this platform. Engagement with datajoint will vary from interactions with a WebApp (with no programming) on the low end to construction of your own database tables on the high end.

The MATLAB datajoint documentation is here:

<https://docs.datajoint.io/matlab/>

And a nice tutorial can be found here:

<https://tutorials.datajoint.io/setting-up/datajoint-matlab.html>

You can read about relational databases here:

<https://en.wikipedia.org/wiki/Relational_database>

And this website has some good information on SQL – much of which is beyond what you need to know, but could be a good reference:

<https://www.w3schools.com/sql/>

What is stored in the database?

The more the better! And this list will certainly grow in time. The short-term plan is for the database to include:

• Every live animal in our colony.

• Deceased animals that have been used in electrophysiology, imaging, or behavior experiments.

• Experimental events for animals, including injections, behavior experiments, etc.

• All electrophysiology data recorded on the rigs in Symphony.

• Stored results from code that has been run on the data.

Longer-term, we will also include

• All cell, retina, and brain images and associated segmentation, counts, and other results.

• Behavioral pose data, audio recordings, and additional analysis results.

• Functional imaging data.

What is not stored in the database?

• Code: while we have lots of code to interact with the database, the code itself will remain stored in GitHub.

• Raw data: As with our current system, the .h5 files that come from Symphony contain the raw data, and we use files that point to them. Similarly, each epoch in the database will contain a link to the associated raw data, but we must keep the original .h5 files. A similar division between raw and processed data will be established for images, behavior videos, and functional imaging data. This will make it so the database does not grow unmanageably large.

Terminology.

• **Table**: the basic unit of the database, not to be confused with a MATLAB table. Tables store entries (e.g. datasets, epochs). Entries in the table must all have the same fields.

• **Field** (also called an **attribute**): a property that belongs to each entry in a table. For example, ‘sample\_rate’ is a field of the Epoch table. It’s similar to a field in a MATLAB struct except, critically, once it is set to a value, it cannot be changed.

• **Schema**: a grouping of tables whose most important purpose is to control permissions. I am managing permissions for users on each schema (described below) in a way that maintains the safety of shared data but allows users control over their own results and such.

• **Query**: a “search phrase” that returns some part of a table. Queries are the most powerful part of relational databases. You can search for any combination of entries you want (e.g. to get only the datasets that match your set of criteria), and queries typically run VERY FAST. I find that baseball statistics are a good way to think about queries, and indeed they are stored in relational databases. For example, you could write a query to find all players from 1975-1982 who batted over .300 left handed and under .250 right handed and hit more than half their home runs with two strikes. Searching the millions of entries in the entire database of major league baseball history for such an esoteric query would take a matter of seconds.

• **Epoch**: a single electrophysiology epoch saved by Symphony – familiar to those in the lab.

• **Dataset:** a grouping of epochs from the same cell as made using the CellDataCurator app – familiar to those in the lab.

• **Pipeline:** a set of data (and a query to find new data) and the associated analysis functions and results. Pipelines are a key concept in the SchwartzLab datajoint implementation, and there is more detail on them below.

Naming conventions.

• **Table names** in datajoint start with a capital letter for each word and have no underscores (e.g. AnimalEventDeceased, PipelineQuery).

• **Field names** are all lowercase and use underscores between words (e.g. epoch\_filter\_func, use\_cell\_id\_list)

Where are all the pieces stored?

• **The database** is stored on a UNIX server in a data center in Evanston that is backed up nightly. Its address is: vfsmdatajoint01.fsm.northwestern.edu. You never need to log into that server directly and your netids do not have permission to do so.

• **Raw data** must be in the RawDataMaster folder on our server:

fsmresfiles.fsm.northwestern.edu/fsmresfiles/Ophthalmology/Research/SchwartzLab/RawDataMaster/. The datajoint UNIX server mounts that directory locally, so after curating your data, *you no longer need copies of the raw data on your own machine*.

• **CellData files** are now an intermediate step between the raw data recorded in Symphony and the database. They store the dataset definitions and spike times while you are creating them in CellDataCurator, but once data is imported into the database, they are no longer used. Nonetheless, we will keep saving them at fsmresfiles.fsm.northwestern.edu/fsmresfiles/Ophthalmology/Research/SchwartzLab/CellDataMaster. If you want to change a dataset definition or spike detection in the database, you can reload either the whole CellData file or the spikes for particular epochs.

• **Common Datajoint MATLAB code** is in this Github repository that you should clone to your own computer: <https://github.com/SchwartzNU/DJ_schwartzlab>

• **Common analysis functions** will be stored at <https://github.com/SchwartzNU/DJ_schwartzlab> and updated frequently by myself and others.

• **Each user’s analysis functions** can be stored on your own computer. I *highly* recommend making your own github repository for them, and I am happy to host it on the lab one. Once a function is working well and someone else might want to use it, it should be copied over to the common analysis functions folders.

• **AnimalInterface WebApp** is running on the datajoint UNIX server and can be accessed via any web browser at <https://schwartzlabdj.fsm.northwestern.edu:9988/webapps/home/session.html?app=AnimalInterface>

DO NOT run the local MATLAB version. This is just for me to make changes and then compile them and copy them over to the web app.

Getting started and connected to the database

• Install the datajoint MATLAB toolbox using the instructions here: <https://tutorials.datajoint.io/setting-up/datajoint-matlab.html>

• Next, you need to clone the repository at <https://github.com/SchwartzNU/DJ_schwartzlab> onto your own computer and put it with all its subfolders on your MATLAB path.

• Now, you can connect with the command: dj.conn.

It is called like this:

dj.conn(server, username, password);

The server is ‘vfsmdatajoint01.fsm.northwestern.edu’

Your username is your first name with a capital first letter (e.g. David, Greg, Sophia). So far we don’t have duplicate first names in the lab, but when we do, I’ll add Last initials for those people.

Your temporary password is Username\_temppwd (e.g. David\_temppwd).

• IMPORTANT: You must be EITHER connected to the GlobalProtect VPN if you are outside the lab and be a member of our VPN group (set up though IT) or be connected within the lab (described below). If you are on our group, in the Settings/Connection tab of GlobalProtect VPN, your assigned local IP will start with 10.120.14. If you are in the lab, you must be connected to an ethernet cable in one of the offices or the main lab. In that case, you should NOT be on the VPN and your IP address (found in your network settings) should begin with 165.124.232. I know it’s a bit of a pain, but restricting access to only these IP addresses is a security measure imposed by IT, and it makes it very difficult for anyone to hack in and corrupt our data.

• Once you log on successfully the first time, use the command:

dj**.**setPassword(‘some\_new\_password’);

to reset your password. Once you have done this, store your hostname, username, and password in your MATLAB startup file by adding lines like this to startup.m.

setenv('DJ\_HOST', 'vfsmdatajoint01.fsm.northwestern.edu');

setenv('DJ\_USER', 'Greg');

setenv('DJ\_PASS', 'greg\_new\_password');

Once you do this. When you start MATLAB again (or just run startup.m), you can connect to the database by simply calling:

dj.conn

• NOTE: You will probably need to reset your password the first time you connect from the VPN AND the first time you connect from inside the lab. Set them to the same thing and then you won’t have to worry about it again.

• If you ever forget your password, just let me know, and I can reset it. I am the database administrator, so I can reset passwords and change permissions for each user. However, I cannot see the password you have entered.

• If you see something that looks like this

>> dj.conn

connection\_id()

+---------------+

2503

ans =

Connection with properties:

host: 'vfsmdatajoint01.fsm.northwestern.edu'

user: 'Greg'

initQuery: ''

use\_tls: 'none'

inTransaction: 0

connId: 0

packages: [0×1 containers.Map]

foreignKeys: [0×0 struct]

isConnected: 1

then you are connected.

Multiple schemas with different permissions.

The schemas are organized as follows. They all begin with ‘sl’ (for SchwartzLab), and you will find a folder for each one (beginning with a ‘+’) in the DJ\_schwartzlab/datajoint-SchwartzLab/ folder.

**sl**

Contents: all the core data that in inserted but not changed. This includes tables like Animal, SymphonyRecordedCell, Epoch, Dataset as well as lookup tables like User, Rig, Genotype, BrainArea, InjectionSubstance, and all AnimalEvent tables (e.g. AnimalEventGenotyped, AnimalEventBrainInjection, AnimalEventReservedForSession) that log when events occur.

Permissions:All users have access to view and insert elements into all tables in **sl** but not to delete or replace elements. I have a master account (not the normal ‘Greg’ account I will use as a lab member) with delete access in **sl** that I can use in case elements are added incorrectly. Insertions into the tables in **sl** occur though the core SchwartzLab datajoint code and should NOT be made by hand with the insert command.

**sl\_mutable**

Contents: tables that store data than will be overwritten. This includes tables like DatasetAnalyzed, which records which datasets have been analyzed in which pipelines and CurrentCellType, which records the type currently assigned to each cell. SpikeTrain is also in **sl\_mutable** so that users can overwrite spike trains for particular epochs in cases where spike detection was not accurate, though most of the (large) SpikeTrain table should remain stable. Please don’t go messing around with it by hand.

Permissions: All users have access to view, insert, replace, and delete elements in **sl\_mutable**, but these interactions will occur through the core datajoint functions and apps. PLEASE DO NOT insert and delete elements in **sl\_mutable** directly with datajoint commands like insert and del.

**sl\_username (e.g. sl\_zach, sl\_devon)**

Contents: whatever you want, sort of. Your own schema starts with 5 tables that you should not delete. EpochResult, DatasetResult, CellResult, and Result, store computed results for epochs, datasets, cells, and multi-cell analyses, respectively. The table TempFilter is a temporary table for user-defined functions that filter epochs. Just ignore it. Your own schema can include any additional tables that you want. You can make them yourself, and they won’t interfere with anyone else’s data.

Permissions: Users have full access to their own **sl\_username** schema and read access (but not insert or delete) into everyone else’s. This means that everyone in the lab can see your results and potentially copy them into their own pipelines for their own analyses, but no one else can overwrite your results. My master account does have full access in case you need help fixing something.

Exploring the tables

You can see (some of) the contents of any table by just typing its name as “schema.table”. Try these commands to see what fields are in each table:

sl.Epoch

sl.Dataset

sl.CellType

sl.AnimalEventAssignCage

sl\_mutable.SpikeTrain

sl\_mutable.CurrentCellType

sl\_greg.DatasetResult

You might notice that some of the fields appear in ALL CAPS while others are all lowercase. The ones in all caps together represent what is called the **primary key** for that entry. That means that a combination of these fields *uniquely specifies* a single entry. In other words, you will not find two entries in the database in which ALL elements of the primary key match. For example, Epochs are specified by the animal and cell to which they belong but also by their epoch\_number. While multiple Epochs have the same value for one or two of these, they never have the same value for all three. The lowercase fields have no such restrictions.

You will also notice that the values for some fields are listed as '=BLOB=', for example, the protocol\_params field of Epoch. This means that it is a data type other than a number or a string. In the case of protocol\_params, it is a MATLAB struct. You can also store vectors, arrays, or cell arrays in fields of the BLOB type. Importantly, while BLOB fields are necessary in many cases for storing the kinds of data we collect, they cannot be used in queries the same way that the numeric and string fields are. In other words, searching for Epochs with stim\_intensity > 300 is easy and fast while searching for Epochs in which protocol\_params.barLength > 500 is MUCH slower.

Simple queries

Now the fun part. What epochs do you want to see? How about ones from cell 060619Ac3 in which the protocol\_name is “Light Step”.

sl.Epoch & 'cell\_id="060619Ac13"' & 'protocol\_name="Light Step"'

The ‘&’ operator as used here has a different meaning than in normal MATLAB syntax. In database parlance it is called the **restriction** operator. But in English, it’s kind of like AND. Notice that I used **single quotes** around each part of the query and **double quotes** around the values that were strings. When checking a number field, omit the double quotes.

sl.Epoch & 'cell\_id="060619Ac13"' & 'epoch\_number=6'

For numbers, you can use inequality operators (e.g. >, <=) and for strings you can use the word “LIKE” and “%” as a wildcard like this:

sl.Epoch & 'cell\_id LIKE "05%20Ac%"'

That query will return all Epochs in the database recorded in May of 2020 on rig A.

Fetch-ing

So far you have been able to see the results of your queries printed to the screen, but you can’t do anything with the data in the sense that you don’t have a MATLAB variable storing things like a list of ‘cell\_ids’ or ‘Rstar\_mean’ numbers. That is where the fetch command comes in. It returns data from your query. It is called on a query result like this:

queryResult = sl.Epoch & 'cell\_id="060619Ac13"' & 'epoch\_number=6'

data\_structure = queryResult.fetch();

Now the regular MATLAB struct data\_structure will contain data from this query result and you can use it however you like in your code with no worries about overwriting or deleting anything in the database itself. There are 5 ways to fetch data. Make sure you use the correct one. This has been one of my most common datajoint coding mistakes.

fetch()

This will return an array of MATLAB structs (one for each item in the query) with **only the primary key** for each item.

fetch(‘field\_name’)

This will return an array of MATLAB structs (one for each item in the query) with **the primary key** **plus field\_name** for each item.

fetch(‘\*’)

This will return an array of MATLAB structs (one for each item in the query) with **all fields** for each item.

fetch1(‘field\_name’)

This only works when your query returns **exactly one item**. Otherwise it will give you an error. It will return a **single value** for the field you request. This might be a number or a string, but if ‘field\_name’ corresponds to a ‘BLOB’ field, it could be any other data structure. For example:

queryResult = sl.Epoch & 'cell\_id="060619Ac13"' & 'epoch\_number=6'

params = queryResult.fetch1(‘protocol\_params’);

Now params contains a whole struct of parameters for this epoch.

fetchn(‘field\_name’)

This returns a list of **only the value of field\_name** for each item in the query. If field\_name corresponds to a numeric field, it will return a vector. If field\_name corresponds to a string or BLOB field, it will return a cell array. For example:

OFFsA = sl\_mutable.CurrentCellType & 'cell\_type = "OFF sustained alpha"';

cell\_list = OFFsA.fetchn('cell\_id')

Now cell\_list is a cell array with the cell\_id of every OFF sustained alpha in the database.

Pipelines**.**

A pipeline is a construct that includes everything you need to go from the database to your finished analysis results (e.g. for a figure in a paper). Pipelines have several associated parts. This section will introduce you to the concept, but it will not give you the tools to manipulate pipelines. I made an App for that that I will introduce in a separate document.

**Owner**

Each pipeline is owned by the user who creates it. This means that the Results for the pipeline are stored in that user’s schema. As mentioned above, however, other users can read the data in anyone’s schema so all pipeline results are readable to all.

**Data query**

This is the query that provides the input to your pipeline. It can be quite complex and include fields from different tables. For example, you could look for Datasets named “Spots Multi Size” with Rstar\_mean = 0 from ON alpha cells or PixON cells from female Animals while excluding Datasets that have the word “\_UV” in their names. The beauty of writing a query as input is that if new data matching the query appears in the database, it will just “show up” in your pipeline, whether or not you recorded it. If you want to narrow your query and exclude or include particular cells or datasets directly by name you can do that too.

**Analysis steps**

This is the list of functions that you want to run on your data and the associated parameter set for each function (if the function has analysis parameters). The functions are run in an order so, for example,

Step1: EpochResult

you can count the spikes in the pre, stim, and post times for each epoch before …

Step2: DatasetResult

using those spike counts to construct a curve of spikes vs. spot size. Then you could compute some value like the suppression index (SI) for each of these curves and …

Step3: CellResult

collect SI for each cell and

Step4: Result

compare the average SI in ON alphas vs. PixONs.

**Results**

Results from your pipeline are tagged with the pipeline name and stored in the EpochResult, DatasetResult, CellResult, and Result tables in your own schema. Storing the results in the database like this means they don’t have to be recomputed. When new data arrives from your data query, you can calculate the results for just that new data. If you update your analysis code, you can choose to run the pipeline and overwrite all the stored results.

**Export settings**

Results from any point in your pipeline can be exported to hdf5 files for plotting in Igor. You can save settings that structure the hierarchy and naming conventions in the hdf5 file. This will keep your Igor experiments well organized and make it easy to create the same kinds of plots for different epochs/datasets/cells with minimal effort. When you run your pipeline, you can choose to have it export the hdf5 files according to your settings or you can run without exporting and manually export the results whenever you like.