Database Solutions for Perovskite Solar Cell Manufacturing

Justin Chu, Lauren Sidarto, Ryan Vo, Long Le

Abstract

The Solar Energy Innovation Laboratory (SOLEIL), under Professor David Fenning, has been manufacturing and collecting perovskite solar cell data at an increasing rate. This data, which comes in many different formats, is hard to store, query, and analyze due to its complexity. The team would carry out their analyses by breaking down the data by sample batch and running analysis code on each, creating a lot of redundancies in their data analysis pipeline. In addition, the raw data itself is currently being sent to a Google Drive folder in a partially unregulated manner. The team hopes to better utilize their data to improve their solar cells - our project seeks to create a database that allows them to record data in a predictable manner, and easily query the data for analysis and visualization purposes. We are looking to use either a Relational Database or a Graph Database approach to clean up and store the team's raw lab data so that they can easily access and query data as needed.

Introduction

Our project group works with Prof. David Fenning, an assistant professor in NanoEngineering at UC San Diego, who runs the Solar Energy Innovation Laboratory. The lab aims to manufacture and test solar cells to improve them in different ways - lifespan, efficiency, and manufacturing consistency, to name a few. The lab is in a phase where they have an established manufacturing process and robust data collection, but lack a data storage and analysis procedure that allows for consistency, accessibility, and scaleup. Our group aims to design and build a database solution that allows the lab to organize their existing data, easily add new cells' data, and efficiently draw insights. Some of the requirements and considerations for this database include:

- Individual, cell-level chemical compositions, linked to traceable batches of chemicals
- Detailed, cell-level manufacturing data (e.g. steps, completion time)
- Cell-level performance metrics, in a variety of data types (e.g. single integers vs. entire tables)
- Fast queries over a large scale of data for visualization and analysis purposes
- Data insertion, edits, and deletion are operations that are less frequent
- A modular structure that allows for future addition of new procedures, steps, or metrics

Our group is currently evaluating the differences between a Graph Database and a traditional Relational Database (RDB). We have hypothesized that a Graph Database might be more suitable due to the lab's need for fast queries; RDBs are theoretically slower due to an O(n) join time in addition to the O(log(n)) lookup time, for both database types[1]. We hope to verify this via testing by the end of this

[1] Gubichev, Andrey. Query Processing and Optimization in Graph Databases. Technische Universitat Munchen, 29 Jan. 2015. pg. 33

quarter, and commit to the more robust solution. Concurrently, our group is working on a parser to clean and format past data, so that it can be migrated into the new database when it is built.

Prior Work

For reference, our group looked at some existing databases to attempt to understand how other people approached storing their data, along with investigating the type of data they held. One of the solutions, "DuraMAT's Data Hub"[2], seemed quite basic; it basically was just an online, open-source repository to upload and share data/documents. While the data is shared and easily accessible, one of the cons to this approach is that the data is uploaded in many formats, ranging from CSVs to image files, to PDFs and others, which makes it difficult to perform analysis and queries as it would require tons of data cleaning and wrangling.

Two of the other databases focused strongly on storing materials and their properties. These databases, "The Materials Project"[3] and "HybriD3"[4], allow for querying material data via several filters, such as a material's chemical composition, properties (thermodynamic/physical properties), etc. This is great for searching for material data, which may be used during the experimentation process to create new solar cell units, or during the analysis process, where some material data may need to be referenced. For our use case, we would ideally want a way to store additional data, such as manufacturing data (steps, etc.), performance metrics, and other information, as mentioned previously.

The last database we referenced was The Perovskite Database[5], which seems to have many of the features we want in our solution, but also would likely require a lot of time to achieve. The Perovskite Database's paper[6] gave us a lot of information about their solution and how ours differs. The issues their database approach solves have a lot of similarities with the problems we're trying to solve, such as making access to analysis and machine learning more available by creating an organized database and including experimental data like performance metrics. The developers mention that they looked at several materials databases like we did, and that while these solutions were a good start towards organizing data, they lacked important data like performance metrics, which are extremely useful for evaluating device performance. In those regards, our goals are similar, but one difference between our solutions is the modularity of the database. From what we have seen their data is tabular; stored in CSVs. We want to investigate RDBs and Graph Databases to see if these solutions are better for our context, where data entered may have varying amounts of steps, materials, and performance metrics between experiments. An additional consideration we are looking into with these Relational/Graph databases is query time, since the lab team hopes to prioritize quick queries for fast analysis and visualization.

Data Description

We aim to find a database solution that fits existing perovskite cell data, which has intrinsic, complex relationships - cells have differing chemical compositions, steps, and metrics. solvents or precursors used in cells are from different manufacturing batches, and manufacturing steps themselves

---

[2] "Duramat Data Hub." Welcome - DuraMAT Data Hub, https://datahub.duramat.org/.

[3] The Materials Project, https://materialsproject.org/.

[4] HybriD3 Database, https://materials.hybrid3.duke.edu/.

[5] The Perovskite Database, https://www.perovskitedatabase.com/home.

[6] Jacobsson, T. Jesper, et al. "An Open-Access Database and Analysis Tool for Perovskite Solar Cells Based on the Fair Data Principles." Nature News, Nature Publishing Group, 13 Dec. 2021, https://www.nature.com/articles/s41560-021-00941-3.

contain a differing number of sub-steps and chemicals. This report provides additional detail on the existing data in the following sections.

The solution we find may be applicable to other lab contexts that require records of complex relationships, with a mix of structure and semi-structured data. Any problems that favor fast queries over fast insertion/deletion/edits may also find our work useful.

Methods

Existing Data Pipeline

SOLEIL has been utilizing a system, PASCAL, to output the lab's manufacturing data. PASCAL:

1. Outputs the sample log json, the run log, the characterization raw data folder, and a dimensionality-reduced single .csv which contains fitted metrics of the raw data.
2. Depending on what the lab operator tasked PASCAL to collect, the raw data is uploaded to the lab's Google Drive. Current metric options from the characterization step are:
    i. PL photostability or PL spectroscopy over time
    ii. Visible light transmission spectroscopy
    iii. PL image
    iv. PL spectroscopy
    v. Brightfield image
    vi. Darkfield image

From this, our group determined we would (1) obtain the existing data, (2) perform data cleaning, (3) choose an appropriate database format, (4) push the clean data to the chosen database, (5) create documents/APIs for users to query data, and finally, (6) create documents/APIs for the users to add more data and maintain the database. In this quarter, we focused on steps 1-4, to ensure that we put ample time into choosing a relevant and useful solution.

Obtaining Existing Data

Our group first retrieved existing data from the lab's Drive folder. Every batch is associated with several output files, but two in particular are relevant to us: the file "maestro_sample_log.json" contains all the information about the process of making the sample solar cell in a single worklist, and the file "fitted_characterization_metrics.csv" contains fitted metrics of the raw data.

Data Cleaning

The existing data was largely in JSON format. We wrote a Python script to process and clean the data, and finally parse it into the format that we plan on using. The structure of the "maestro_sample_log.json" JSON file is as follows:

```
 {"sample0": {"name": str, "storage_slot": dict, "substrate":
str, "worklist": list, "hotplate_slot": dict} , "sample1":
{...}, ...}
```

The following are some observations regarding the data:

- The list in the `worklist` field has the manufacturing steps, in order of operation. Some steps, such as `characterization_task`, for staging to take a characterization image, and `destination`, for moving the sample, are very standardized and have a fixed format. Because of this, we can easily flatten the nested dictionary into a list.
- For steps that do not have fixed steps (such as `drops`), we do not have a fixed length because each solution might have many different `solvents` and `solutes`.
- `solutions` can have `solutes` and a `solvent`, or just one `solvent`.
    - `solutes` are typically stored in a long string format, for example

      `"FA0.78_MA0.1_Cs0.12_Pb1.09_I2.7_Br0.491_Cl0.0818"`

      which indicates all chemicals and their corresponding concentrations that are solutes. The first entry for solutes (the solutes that are part of the first drop step) are mixed by hand, outside of the PASCAL machine.
    - `solvent` is stored similarly. If a `solution` has an empty string for `solutes`, and an entry for `solvent`, this implies that the `solvent` is actually an antisolvent. The antisolvent gets dropped in at a later date.

The varying number of steps, chemicals, outputs, and properties led our group to consider a number of different database solutions.

Choosing a Database

There were two database types that we considered: a traditional relational database (RDB), and a graph database. After discussing with members of the lab team, we noted that the primary requirements for the database were to:

1) Store the worklist of manufacturing steps, as well as all relevant properties and measurements for each step, ideally with the nominal and actual values, and
2) Keep track of all chemicals involved, how they change through the manufacturing process, and ideally relate them to a batch number
3) Be easily queryable; it should be easy to query for individual solar cell worklists (e.g. fetch all info associated with sample 0 from batch 1), query by a property (e.g. fetch all spin steps with an rpm of 200), and ideally query based on the cell's associated properties (measured after they are manufactured)

Considering these requirements, we chose to start with a relational database, as the group was more familiar with these. This would also help us to familiarize ourselves with the data quicker.

Relational Database

We started by creating a basic schema, as follows:

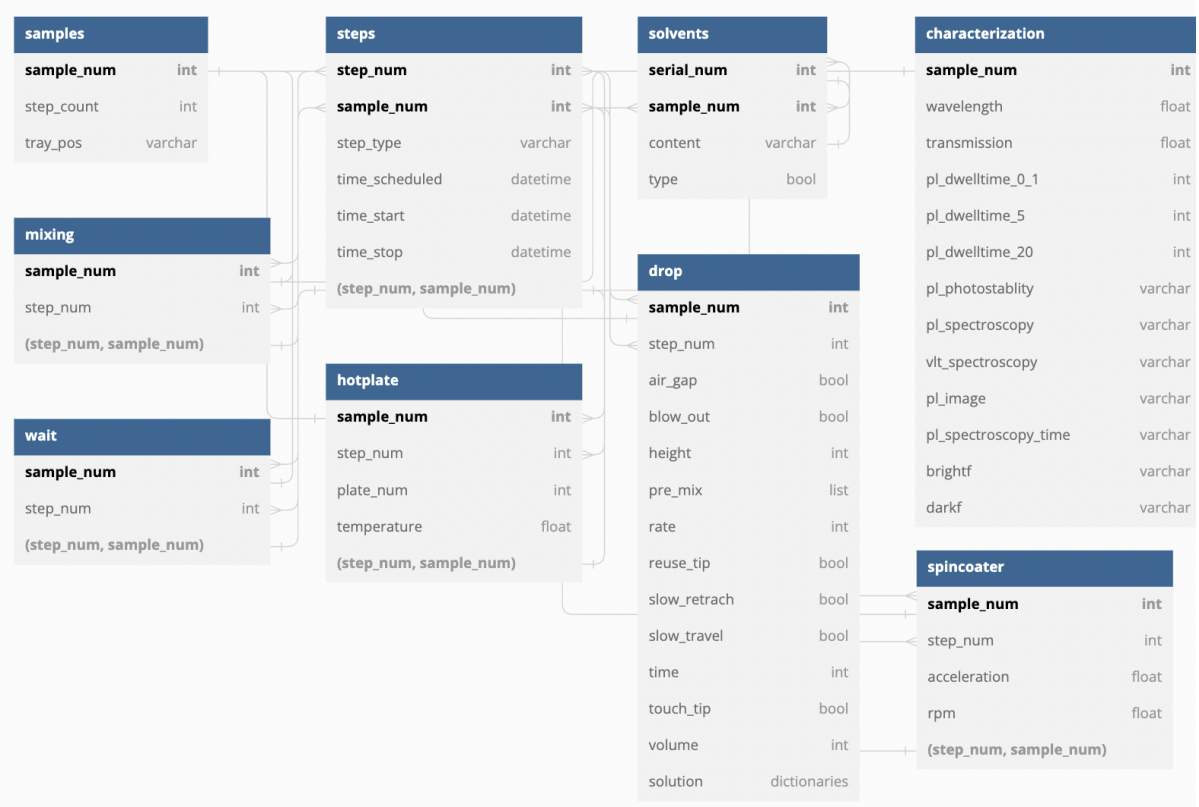

Figure 1: Initial RDB Schema Mockup

In the above schema, every row is a column, and all bolded rows indicate primary keys. Tables with multiple bolded rows indicate compound primary keys. This database keeps track of the worklist by assigning each step a step_num.

The schema fails to take into account different permutations of the same step - e.g. if a cell has multiple `hotplate` steps, there will be a non-unique primary key. A lack of understanding of the data led to this initial oversight.

The `characterization` table is also a placeholder - it was intended to hold characterization outputs, but every row in that table turned out to require a different storage method. Pl_spectroscopy, for example, is a csv file with wavelengths and transmission, and can't be fit into the same table as the other metrics. There is also a difference between the characterization steps themselves (which involve things like exposure time), and and the characterization outputs/metrics; both need to be recorded.

This table also fails requirement (2), since it puts all chemicals in the `solvents` table. Solvents, solutes, solutions, and anti-solvents all have different properties, which are not stored in an intuitive manner here. After continuing to troubleshoot, this schema evolved into the one below:

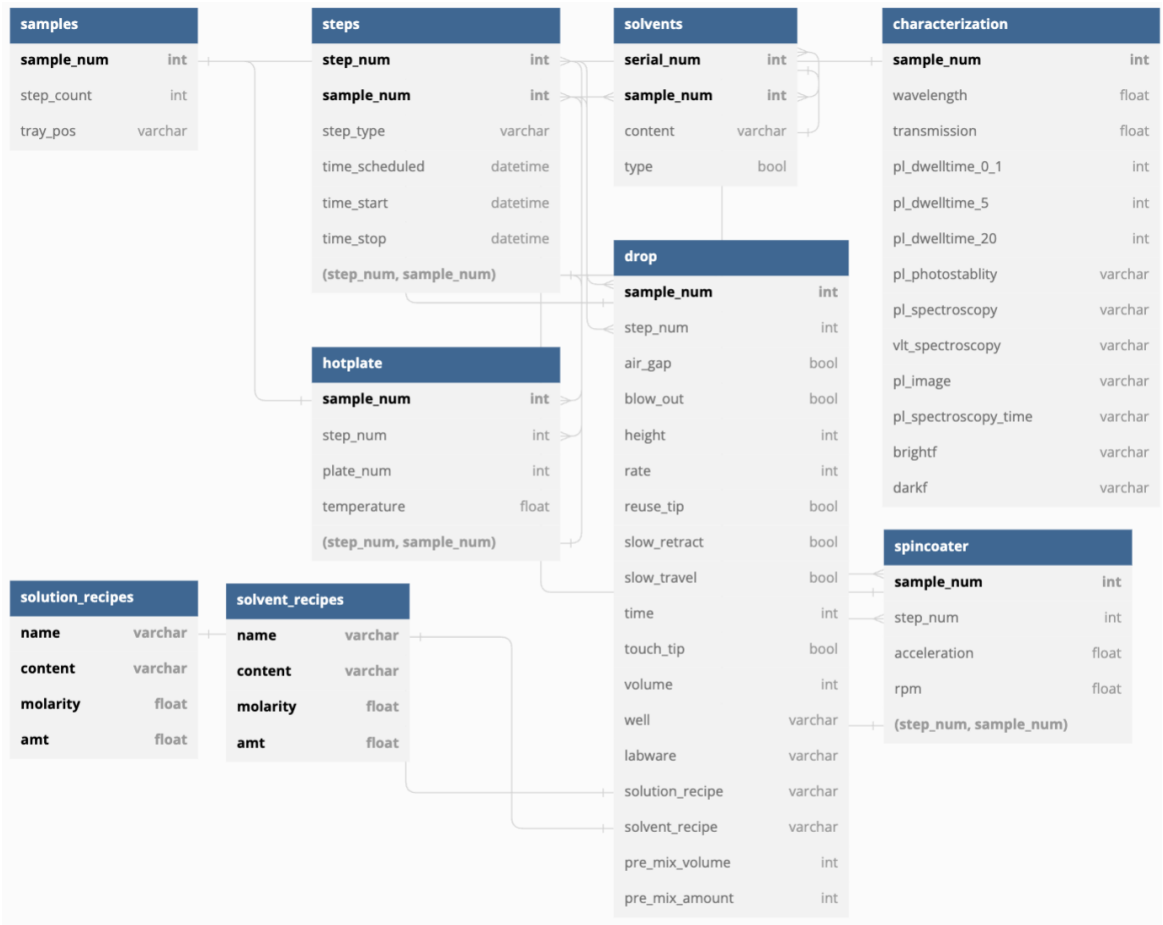

Figure 2: Updated RDB Schema Mockup

This schema has new `solution_recipes` and `solvent_recipes` tables, to keep track of the different chemicals and mix compositions used in each drop step. This is also any RDB solution's largest limitation - in order for a primary key to exist for chemical data, all four columns must be primary keys. The lab tries different combinations of chemicals; only one of those columns might change from experiment to experiment, leaving the other 3 the same.

At this point we conclude that the RDB solution is unavoidably clunky - it would be near impossible to avoid storing duplicate data, especially with the amount of combinations the lab intends to try; it will be inflexible when the lab needs to add new properties and new steps; and, in addition, would have extremely slow query times - as previously mentioned every table would need to be joined (O(n)), and then queried (O(logn) minimum), every time one query is run.

We then sought direction from someone affiliated with Fenning but working in another lab - Rishi Kumar - but attempting to solve the same problem. He talked us through both his own graph database attempt, and another lab's tabular database solution. We decided to spend the quarter replicating both of these.

Tabular Database

Kumar explained another tabular solution he knew of - it involved creating a single table, where every row is a different cell, with a "JSON blob" column that contains the cell's entire worklist in JSON

format. This removes the need for multiple tables, joins, and complex, step-specific columns. This would work for analysis as the file could then be imported into some relational database management system, specifically PostgreSQL, which allows inline JSON queries. The data can then be converted or extrapolated into a different format for analysis, like a pandas DataFrame.

In our replication of this solution, we produced a table of one column with every row as a sample by writing a script that extracted out specific characteristics from the JSON worklist file. This solution is robust because the existing steps do not get affected when a new process is added because the new script will only interact with the new data. This solution keeps the different processes separated, which allows for forward and backward compatibility. Below is an example of our script's output:

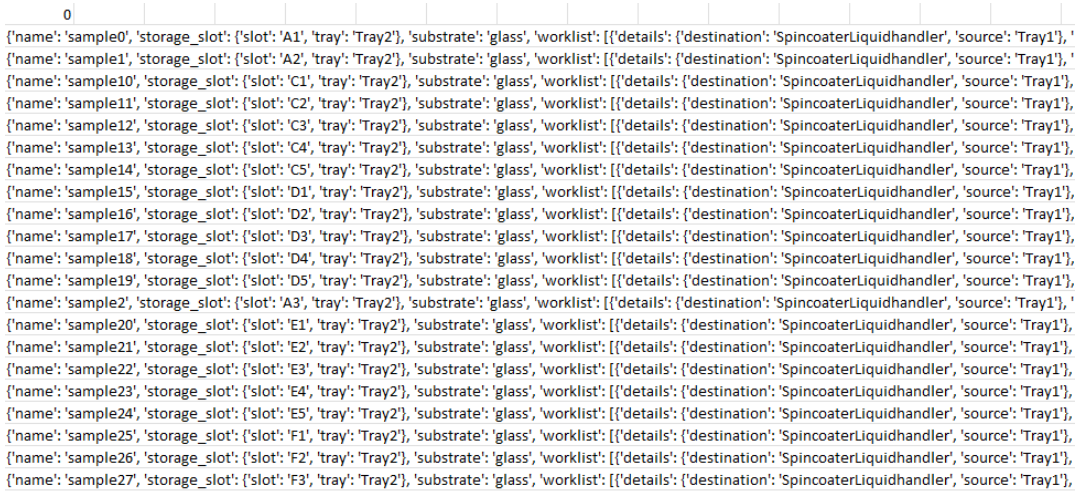

Figure 3: Tabular Worklist Solution

While this works, this solution isn't the most intuitive to use or read. The main issue with this solution is the scalability of the database. For a database with few updates, this solution works well because the user does not need to frequently update the csv files, but, for the SOLEIL lab, this solution is not user friendly because the entire database (with every single csv file) would need to be updated when a new row of a JSON object is added to the main table. Due to this, our group chose not to dedicate time importing this into an SQL system - our time would be better spent looking into other solutions.

To complete the proof of concept, however, we wrote a script that can unravel the JSON object inside each row, extracting attributes such as steps or chemicals. The Figure 4 below depicts the extracted recipe of anti-solvents from the JSON:

sample_num	step_num	molarity	antisolvent	labware_v	labware_t	well	blow_out	height	pre_mix	rate	reuse_tip	slow_trac	slow_trav	time	touch_tip	volumn
20220523_3A3X_Xu-recipe_sample1	2	0	MethylAcetate0.5_WideTip0.5	15mL	Tray1	C1	FALSE	0.5	[3, 100]	450	TRUE	TRUE	TRUE	20	FALSE	75
PbX2 Loading Optimization_sample5	2	0	MethylAcetate	15mL	Tray1	C1	TRUE	2	[5, 100]	80	TRUE	TRUE	TRUE	33	FALSE	100
PbX2 Loading Optimization_sample3	2	0	MethylAcetate	15mL	Tray1	C1	TRUE	2	[5, 100]	80	TRUE	TRUE	TRUE	33	FALSE	100
20220516_3A3X_halfcells_sample30	14	0	MethylAcetate	15mL	Tray1	C1	FALSE	2	[3, 100]	100	TRUE	TRUE	TRUE	45	FALSE	75
20220510_3xHalide_MACl_films_sample9	2	0	MethylAcetate	15mL	Tray1	C1	FALSE	2	[3, 100]	100	TRUE	TRUE	TRUE	45	FALSE	75
WBG Molecular Sieve Dried Solvent_sample4	2	0	MethylAcetate	4mL	Tray1	D1	TRUE	2	[5, 100]	80	TRUE	TRUE	TRUE	33	FALSE	100
20220504_PIN_Half_Cells_sample26	8	0	MethylAcetate	15mL	Tray1	C1	FALSE	2	[3, 100]	100	TRUE	TRUE	TRUE	45	FALSE	75
20220517_3A3X_AS_tip_optimization_sample10	2	0	MethylAcetate0.5_StockTip0.5	15mL	Tray1	C2	FALSE	2	[3, 100]	150	TRUE	TRUE	TRUE	40	FALSE	75
PbX2 Loading Optimization_sample35	2	0	MethylAcetate	15mL	Tray1	C1	TRUE	2	[5, 100]	120	TRUE	TRUE	TRUE	33	FALSE	100
B9-psk_sample1	2	0	MethylAcetate	15mL	Tray1	C4	FALSE	0.5	[3, 100]	450	TRUE	TRUE	TRUE	30	FALSE	75

Figure 4: Tabular Solution: Extracted Anti-Solvents

Similarly, the figure below is the extracted individual chemical drops from the JSON rows:

sample_num	step_num	duration	start_times	steps_acceler	steps_duration	steps_rpm	id	precedent	start	solvent	antisolven
PbX2 Loading Optimization_sample24	2	115	[1, 6.0]	200	5	1000	spincoat-790f679c-30e7-4e40-91c6-1	storage_to_spincoater--73deef78-ff46-4ac1-b	1783	TRUE	TRUE
20220523_3A3X_Xu-recipe-prerun_sample4	2	106	[2]	2000	50	5000	spincoat-a6c40559-0036-4b05-ba0b-	storage_to_spincoater--c9e0faa0-030b-4249-b	697	TRUE	TRUE
20220516_3A3X_halfcells_sample26	8	69	[5]	500	30	3000	spincoat-3484b365-25b2-404b-b52d-	storage_to_spincoater--bd8d057a-ade2-4615-	14721	TRUE	FALSE
20220516_3A3X_halfcells_sample7	2	69	[5]	500	30	3000	spincoat-3e0f631f-79e2-48b2-abc8-	storage_to_spincoater--504ca84a-9259-42f1-8	10226	TRUE	FALSE
20220504_PIN_Half_Cells_sample29	8	104	[1]	2000	50	5000	spincoat-9b4df2fa-9ab7-4059-92e1-	storage_to_spincoater--0b052c28-79c3-4797-9	3288	TRUE	TRUE
20220516_3A3X_halfcells_sample18	14	106	[2]	2000	50	5000	spincoat-c1f7d4e1-9b84-4c9e-abc0-	storage_to_spincoater--ffb05b90-8ecf-42df-a	19092	TRUE	TRUE
20220516_3A3X_halfcells_sample15	14	69	[5]	500	30	3000	spincoat-e7c34ce5-2144-477e-a80a-	storage_to_spincoater--41a676d9-ec6f-4719-b	5931	TRUE	FALSE
PbX2 Loading Optimization_sample34	2	115	[1, 6.0]	200	5	1000	spincoat-a381721f-7a9e-45a8-9fe8-3	storage_to_spincoater--2e701c2b-6020-4d8a-	5728	TRUE	TRUE
20220523_3A3X_Xu-recipe_sample1	2	106	[2]	2000	50	5000	spincoat-cdd96d97-ef95-4ffa-a5d2-(	storage_to_spincoater--1f4316c1-64bd-4b02-b	1195	TRUE	TRUE
20220516_3A3X_halfcells_sample7	20	49	[0]	500	30	3000	spincoat-865643b4-144f-4d09-b40d-	storage_to_spincoater--b6d22512-3eab-432a-	13047	TRUE	FALSE

Figure 5: Tabular Solution: Extracted Drop Steps

Other queries could be written or individual tables can then be joined to achieve the necessary results.

Graph Database

We also attempted to replicate Kumar's graph database solution. He intended to use MongoDB to implement a graph database from scratch, where each node is a document that contains a list of all its predecessors and successors. Kumar notes that a limitation of this solution is that documents have a 16MB size limit, meaning some characterization output files can't be stored within the database.

Kumar's solution, importantly, gave us direction on how to store the data. His solution involved creating a node type for each chemical and action/processing step. `Chemical` nodes would be paired with an action (e.g. a `dissolve` step, indicating the chemical was dissolved into a solution), and the output solution would be stored as a different `Chemical` node. A recreation of this structure is as follows, where gray nodes are `Chemical` nodes, and blue nodes are `Action` nodes:

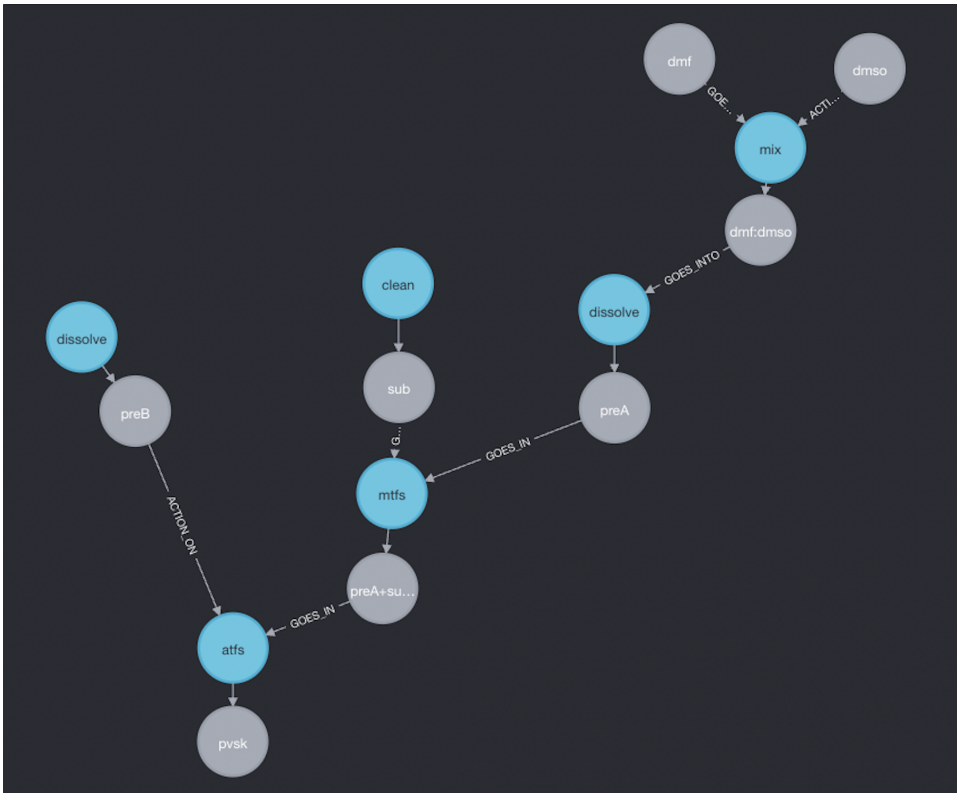

Figure 6: Example Single Solar Cell Worklist in Graph Format

Prior to this our group looked into different graph database providers - we settled on Neo4j, an open-source, NoSQL native graph database management system. Neo4j implements the true graph all the

way down to the storage level instead of an abstraction, meaning that queries are in constant time (O(1)).[7] This fits our use case well - we agreed with Kumar that both teams would look into Neo4j.

In order to replicate Kumar's solution in Neo4j, we first had to look into how to import the lab's data into Neo4j. Neo4j's graph query language, Cypher, has a "LOAD CSV" command that allows us to import the data in order to create the nodes and links from Kumar's solution. The command can load data into Neo4j from a file or a URL. We looked into Google Drive to host files, allowing us to load in files with URLs and allowing backward compatibility with the lab's existing data pipeline. Due to rate limits with Google Drive, we then switched to hosting CSVs on Neo4j Docker directly. This allowed us to create a script to run all the Cypher queries needed to both import the data and create the graph's nodes and links.

After determining what structure Neo4j requires files to use in order to read them properly, we then modified and created data parser functions to convert the JSON worklists into CSVs. To create the nodes and links for the graph representation of the data, we need to parse the JSON into 3 separate CSVs. The first CSV contains the information on all of the chemicals used in the process. This includes things like solutes, solvents, anti- solvents, and mixes of these chemicals. The second CSV contains information on the Action steps of the process, such as `Dissolve, Drop, or Spin` steps (among others). The last CSV contains the information on how to link all of these action and chemical nodes. The first two CSVs would be used to create nodes, and the latter would be used to create relationships. Below is an example preview of these three files for a single solar cell sample:

	chemical_id	content	concentration	molarity	volume	type	step_id
0	1	FA	0.7800	NaN	NaN	solute	1.0
1	2	MA	0.1000	NaN	NaN	solute	2.0
2	3	Cs	0.1200	NaN	NaN	solute	3.0
3	4	Pb	1.0900	NaN	NaN	solute	4.0
4	5	I	2.7000	NaN	NaN	solute	5.0

Figure 7: `Chemical` **CSV**

	step_id	action	chemical_from	drop_air_gap	drop_blow_out	drop_height	drop_rate	spin_acc	spin_rpm	hp_duration	hp_temp	rest_duration
0	1	dissolve	1.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	2	dissolve	2.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	3	dissolve	3.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	4	dissolve	4.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	5	dissolve	5.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Figure 8: `Action` **CSV**

	step_id	action	chemical_from	step_to	chemical_to	step_from	sample_id	batch_id
0	1	GOES_INTO	1	1	<NA>	<NA>	sample0	WBG Repeat, Batch 4 (Experiment 1)
1	2	GOES_INTO	2	2	<NA>	<NA>	sample0	WBG Repeat, Batch 4 (Experiment 1)
2	3	GOES_INTO	3	3	<NA>	<NA>	sample0	WBG Repeat, Batch 4 (Experiment 1)
3	4	GOES_INTO	4	4	<NA>	<NA>	sample0	WBG Repeat, Batch 4 (Experiment 1)
4	5	GOES_INTO	5	5	<NA>	<NA>	sample0	WBG Repeat, Batch 4 (Experiment 1)
5	6	GOES_INTO	6	6	<NA>	<NA>	sample0	WBG Repeat, Batch 4 (Experiment 1)

[7] "What Is a Graph Database? - Developer Guides." *Neo4j Graph Data Platform*, https://neo4j.com/developer/graph-database/.

Figure 9: `Links` **CSV**

All of these CSVs are connected via `step_ids` and `chemical_ids` which allows for the necessary creation and linking of nodes in Neo4j. These graphs and their nodes have a `sample_id` and `batch_id` which helps ensure that each sample's graph is separate but also queryable via these identifiers (e.g. in **Figures 7, 8, and 9** above, sample 0 comes from Batch 4, Experiment 1. Each batch has around 40 samples). **Figure 7** contains some of the chemicals, which are dissolved (**Figure 8**) in steps 1-6 (**Figure 9**). Not pictured are rows for the characterization output nodes - nodes that store the actual output of metric-recording characterization steps. Since Neo4j only supports primitive data types in node properties, we transformed images into lists and tables into dictionaries, which can easily be turned back into their original format using Python.

Once this was completed, moved on to creating the graphs for a batch of samples. In addition to the ability to import data, the "LOAD CSV" command also allows for the creation and linking of nodes via a query that is attached to a load command. An example of the Cypher code used to accomplish this is shown below, where a CSV is loaded and used to create a `Chemical` node with respective properties:

```
LOAD CSV WITH HEADERS FROM
"https://drive.google.com/uc?export=view&id=[fileid]" AS row
CREATE (c:Chemical {chemical_id: row['chemical_id'],batch_id:
row['batch_id'], content: row['content'], molarity: row['molarity'],
concentration: row['concentration'], volume: row['volume'], type:
row['type'], sample_id: row['sample_id']})
```

Unfortunately, Cypher LOAD commands are unable to span multiple queries. This meant that for a full graph, we would run 6 LOAD CSV queries in order to create and link all of the nodes. This is inefficient for one sample, let alone for 40+ samples. Thankfully, Cypher allows for running scripts via their Cypher shell terminal in order to run multiple queries at once. This required us to save the queries to a .cypher file where each query/LOAD CSV command is semi-colon separated. To make use of this, we created a script to scrape the file IDs for the CSVs in Google Drive folder into a spreadsheet. Then, we wrote functions to generate the necessary queries to create a graph for each of the samples in the batch and wrote them to a .cypher file, allowing us to automate the creation of these graphs.

We noticed that Neo4j Docker had quite a few limitations in terms of use cases and the ability of the version of Neo4j, so we decided on trying to use Neo4j Desktop, as it gave us the possibility to complete the objective of using python to use Neo4j. Once we were able to create the graph in Neo4j, Desktop, we looked into using Python to run Neo4j, because the lab members were more familiar with JupyterHub and Python. The solution that we found was a python package called py2neo, which allows for querying the database in python and working with the data without requiring any data exports. The database storage and access approach we came up with with the lab group was to use a host computer that would act as a server where the people in the lab could access and use the database on JupyterHub.

Results

After graphing the nodes and links from our cleaned/transformed data, we are able to generate a graph for each sample. The figure below shows the contents of one sample:

Figure 10: Graph for Individual Sample

The blue `Chemical` nodes are connected to red `Action` nodes such as dissolve or drop. The worklist for this particular cell involves hand-mixing several chemicals into "Mix1", which is then fed into a machine, where the rest of the steps take place. An antisolvent, MethylAcetate, is also fed into the machine; both the "Mix1" solution and the antisolvent are then dropped together to form "Mix2". The new solution then undergoes a series of spin coater, hotplate, rest, and characterization (measurement) steps. The sample is moved to "tray2" at the end, its final resting state. All characterization steps also have another `Char_output` node, which stores the actual output of the measurement (images, readings, etc.). After running the all queries via our graph script, the database contains all the samples from one batch:

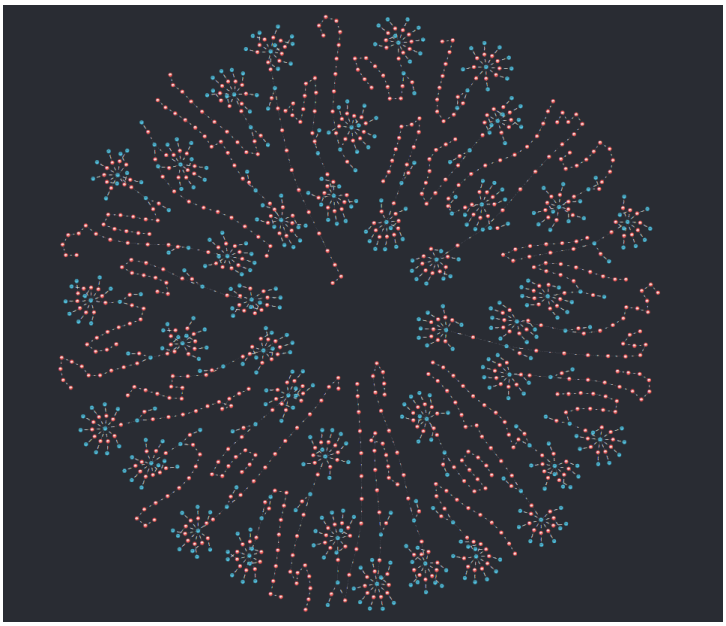

Figure 11: Graph Database with all Batch1 Samples

The query time of the graph with more and more samples is displayed in Figure 12, where we compared the following queries:

```
One Node Per Sample
MATCH (n) WHERE(n.anneal_temperature = '110.0') return n
Multiple Nodes Per Sample
MATCH(n) WHERE(n.drop_height = 0.5) return n UNION ALL MATCH(n)
WHERE(n.anneal_temperature = '100.0') return n
```

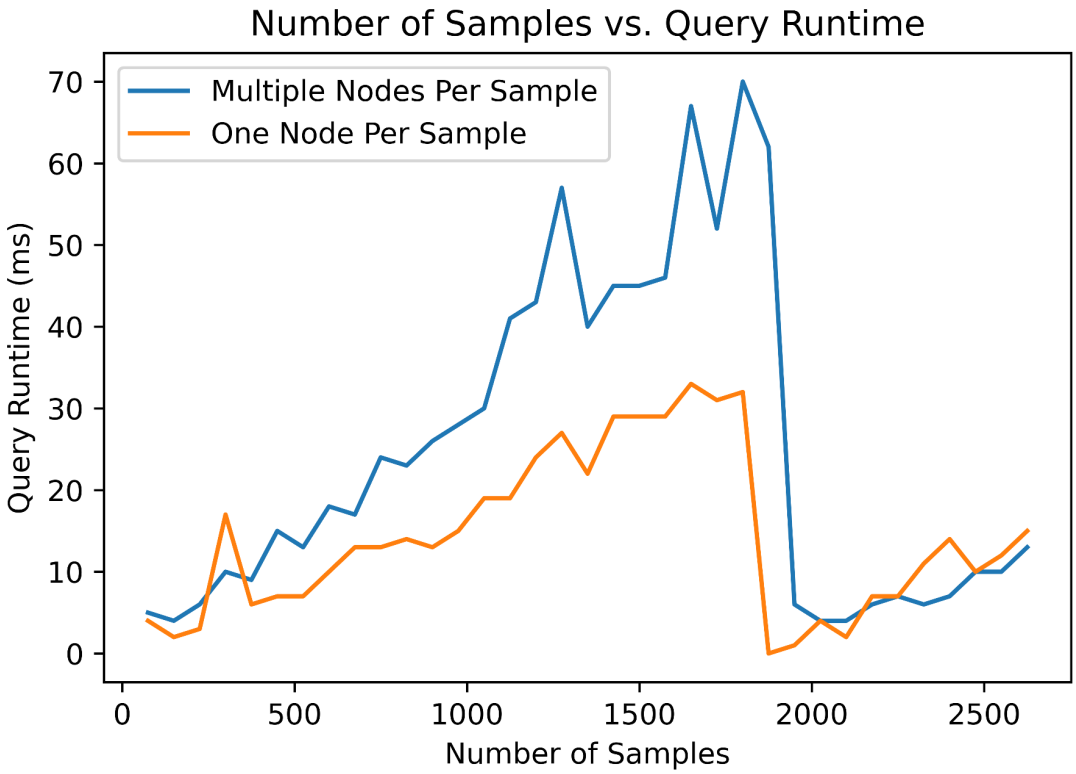

Figure 12: Query Runtimes with Increasing Number of Samples

We also queried the entire graph in Neo4j and ran the equivalent in the RDB solution and found that Neo4j was significantly faster than the RDB solution. The RDB solution takes $O(n^6)$ time because it has to join 6 different tables per batch, leading to exponential growth on runtime for querying. Below is the graph showing the runtimes on a growing number of samples for both Neo4j and the tabular database.

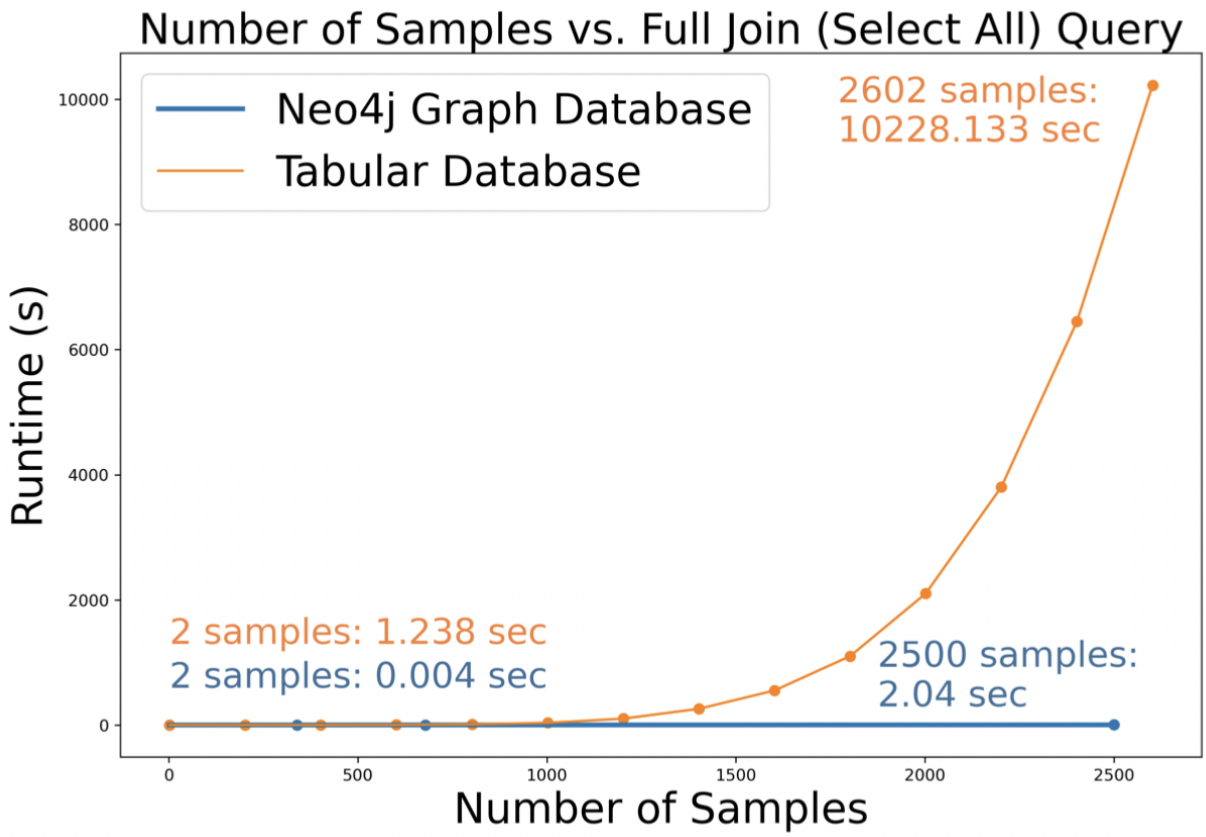

Figure 13: Query All Runtimes Neo4j vs. RDB Solution

Conclusion

Our group implemented a working lab-to-database pipeline by replicating some existing ideas. In our discussions with Kumar and prior research, we found no similar solutions exist.

The usefulness of this solution is largely derived from Neo4j's Cypher - queries are relatively intuitive and simple to write, though we would need to communicate more with Fenning's data team to see if there are necessary actions that Cypher does not support.

Example Queries

The database can be queried and filtered for features of interest like sample_id, temperature variables, or chemical content. Corresponding samples, nodes, or relationships, can be returned. For example, the following query returns the graph of one sample, similar to **Figure 10**:

```
MATCH (n) WHERE n.sample_id="sample0" RETURN (n)
```

This query allows us to find nodes that have a specific property, like a rate of 80:

```
MATCH (n:Action) WHERE n.rate=80 RETURN (n)
```

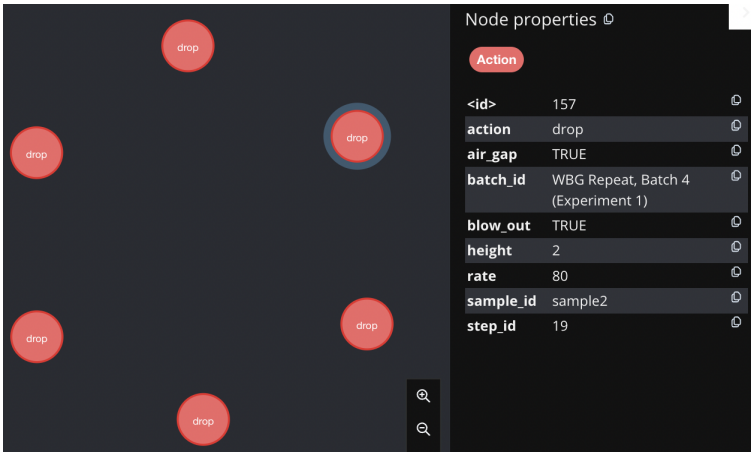

Figure 14: Node Query Output

And this query allows us to find all Br chemicals, as well as how they're dissolved:

```
MATCH (n {content: 'Br'})-[r]->(c) RETURN *
```

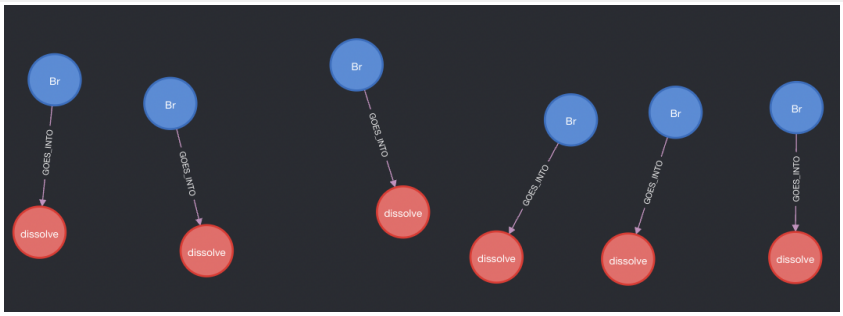

Figure 15: Node and Relationship Query Output

Limitations

The Methods section details how we have since solved two of the main limitations we had - the Google Drive rate limit, and the storage of characterization outputs. Our current challenge is working out hosting - Neo4j Desktop only runs locally as far as we are aware, and other versions of Neo4j with managed databases are paid. We decided to put the database on their server computer so that the database would be exposed and they would be able to use Jupyter Notebooks to access the database via the aforementioned py2neo package.

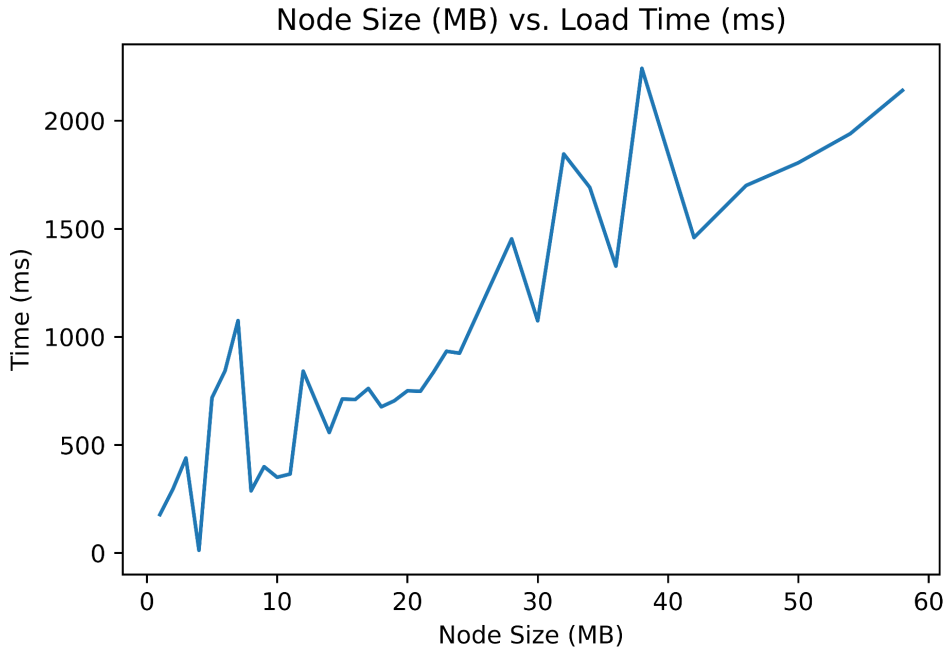

Figure 16: The Time it Takes to Load Nodes

The figure above shows the time it takes for Neo4j Desktop to load in nodes from a csv by the size of the node. The limitation on Neo4j Desktop for importing the nodes is the heap size, which can be increased on the platform itself.

There are also further financial and technical limitations with Neo4j itself, which has different tiers, pricing, memory limits, and other constraints we have yet to run into. Further research is needed; we detail this below.

Future Work

Our group has also determined some courses of action we would ideally take from here - there are several aspects of this solution that need to be fleshed out further:

An important step moving forward would be to meet with the lab team to talk about the best way to output "workable" data moving forward, so that at the end of our data pipeline they can run any queries or analyses they might need easily.

Last but not least, we would need to look further into Neo4j for the future. Assuming this is a model that will be adopted and used for a while, we need to investigate pricing, size limits, and query effectiveness/efficiency as the database grows in size. We also would like to become more familiar with their Cypher query language to understand not only its limitations but also its potential for how complex queries can be.

References

1. Gubichev, Andrey. Query Processing and Optimization in Graph Databases. Technische Universitat Munchen, 29 Jan. 2015. pg. 33

2. "Duramat Data Hub." Welcome - DuraMAT Data Hub, https://datahub.duramat.org/.

3. The Materials Project, https://materialsproject.org/.

4. HybriD3 Database, https://materials.hybrid3.duke.edu/.

5. The Perovskite Database, https://www.perovskitedatabase.com/home.

6. Jacobsson, T. Jesper, et al. “An Open-Access Database and Analysis Tool for Perovskite Solar Cells Based on the Fair Data Principles.” Nature News, Nature Publishing Group, 13 Dec. 2021, https://www.nature.com/articles/s41560-021-00941-3.

7. “What Is a Graph Database? - Developer Guides.” Neo4j Graph Data Platform, https://neo4j.com/developer/graph-database/.