

UNIVERSITY OF TARTU  
Institute of Computer Science

Heidi Carolina Martinsaari, Kertu Nurmberg,  
Vera Akinyi Onunda, Mart Traagel

# **Data Engineering Report**

## **Group 2**

**Supervisors: Ahmed Mahmoud Hany Aly Awad,**  
**Riccardo Tommasini**

Tartu 2022

# Contents

## [Contents](#)

## [Project and Source](#)

### [1.1. Goals of the project](#)

## [2. Database models](#)

### [2.1 Feature selection](#)

### [2.2 Relational database](#)

### [2.3 Graph database](#)

## [3. Pipelines](#)

### [3.1 Creating tables for relational database](#)

### [3.2 Importing data from source to staging area and cleansing](#)

### [3.3 Splitting data for the relational database and importing data to PostgreSQL.](#)

### [3.4 Enriching data using same source](#)

### [3.5 Importing data to Neo4j](#)

## [4. Analysis](#)

### [4.1 Validation of relational database](#)

### [4.2 Analysis in the relational database](#)

#### [4.2.1 Parents, children, grandchildren, ...](#)

#### [4.2.2 Most popular tags and tags related](#)

#### [4.2.3 Timeline of the added memes by origins](#)

#### [4.2.4 Analysing links](#)

### [4.3 Analysis in the graph database](#)

## [5. Teamwork](#)

### [5.1 Initial plan](#)

### [5.2 Final outcome](#)

## [6. Lessons learned](#)

### [6.1 Challenges constructing pipelines](#)

### [6.2 Challenges building relational db](#)

### [6.3 Challenges creating graph db](#)

## [Summary](#)

## [Extra 1. Schema of the relational database in PostgreSQL.](#)

# 1. Project and Source

## 1.1. Goals of the project

The goal of the project was to model relational and graph databases and extract, transform and load the given dataset to these databases. For the starting point it was provided a source database in the format of a json file which could be downloaded from the [link](#). The data consisted of internet memes and related information. There were also two sets of enrichment: [Google Vision Enrichment](#) and [DBPedia Spotlight Enrichment](#).

ETL processes had to be designed and implemented as Apache Airflow pipelines. We built five separate pipelines: the pipeline which loads the data from source and does the cleansing, the pipeline which creates the relational database tables, two pipelines that shape the data and load it to both databases - into relational database and graph database and the pipeline for enriching the relational database.

After building and populating the databases we did some query-based analysis. Actually our work was kind of iterative. After the first successful population we discovered many weaknesses of our solutions and we made many fixes to the databases. In the next analysis part we discovered more problems to solve and even now we have many more ideas how to enhance the databases.

## 2. Database models

### 2.1 Feature selection

Before starting any modeling we explored the data and then decided which features to select. We made a selection mostly based on our interests and what we thought to be important for describing memes, but we also left out features which did not give any additional value like repeating values. The file for feature selection was maintained [here](#). In addition to feature selection we decided to import only the memes (where category was equal to Meme).

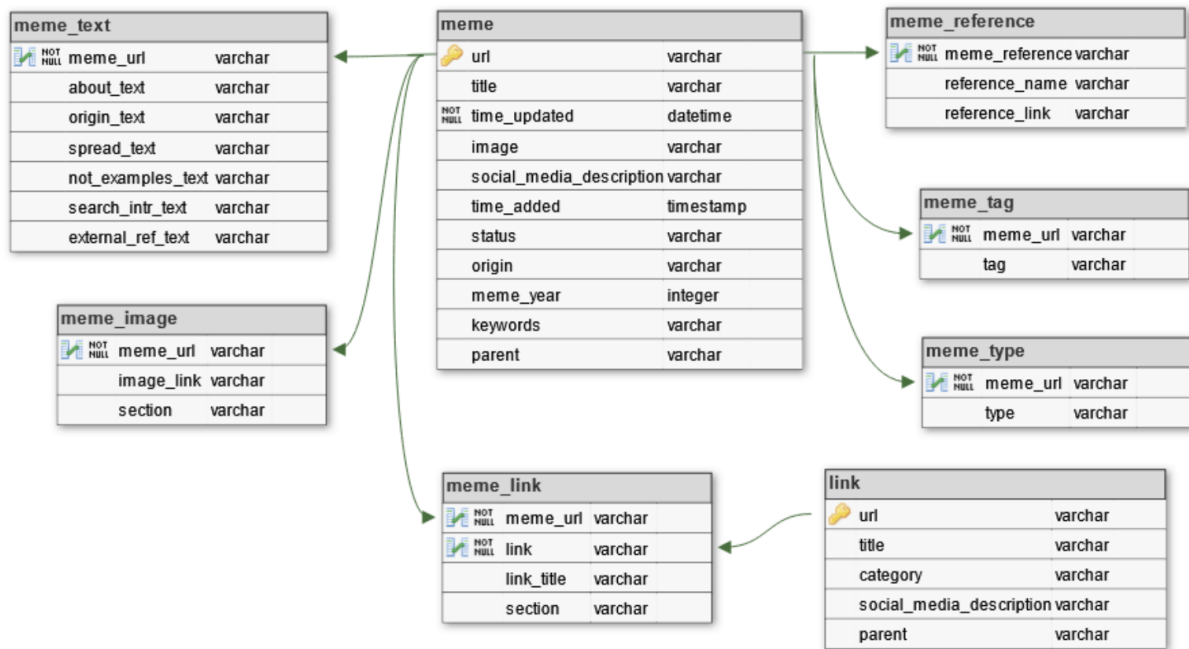
### 2.2 Relational database

As described in the first section we enhanced the model in many iterations. In the first phase we had a monotable which contained unflattened data fields, e.g tags. All the tags were listed in the same field. All the lists were unique for each meme. It was almost impossible to query such fields and make analysis. We decided to split data into multiple tables.

The first idea for splitting was to find the fact and dimension tables, but it must be indicated that the topic of internet memes is very challenging. The memes data differs from usual business data which would have been more familiar for most of us. Thus, we splitted the data into *objects*, like meme, link, tag, external reference, etc, and *relationships* like meme\_tag, meme\_reference.

We also had such relationships as meme\_about\_link, meme\_origin\_link, etc, but in the next phase we discovered that these were repeating each other, so we joined these tables together as meme\_link, the relationship between meme and link.

Actually, the tag and external link table we did not create as we did not see them necessary, but we created a link table within the enrichment task. We loaded more data for describing links from the initial source file as the links we had in about link, origin link, etc were other categories we did not choose (only memes). Some links were also not in the source file. So, we derived their information based on the link and with the help of natural language analysis.



**Figure 1.** Schema of the relational database in PostgreSQL.

Above you can see the graph of our relational database. The table meme is the ‘main’ table as the memes were the centre topic of the dataset. Hereby, tables meme and meme\_text are the same table but meme\_text is separated from other meme features because of long text fields.

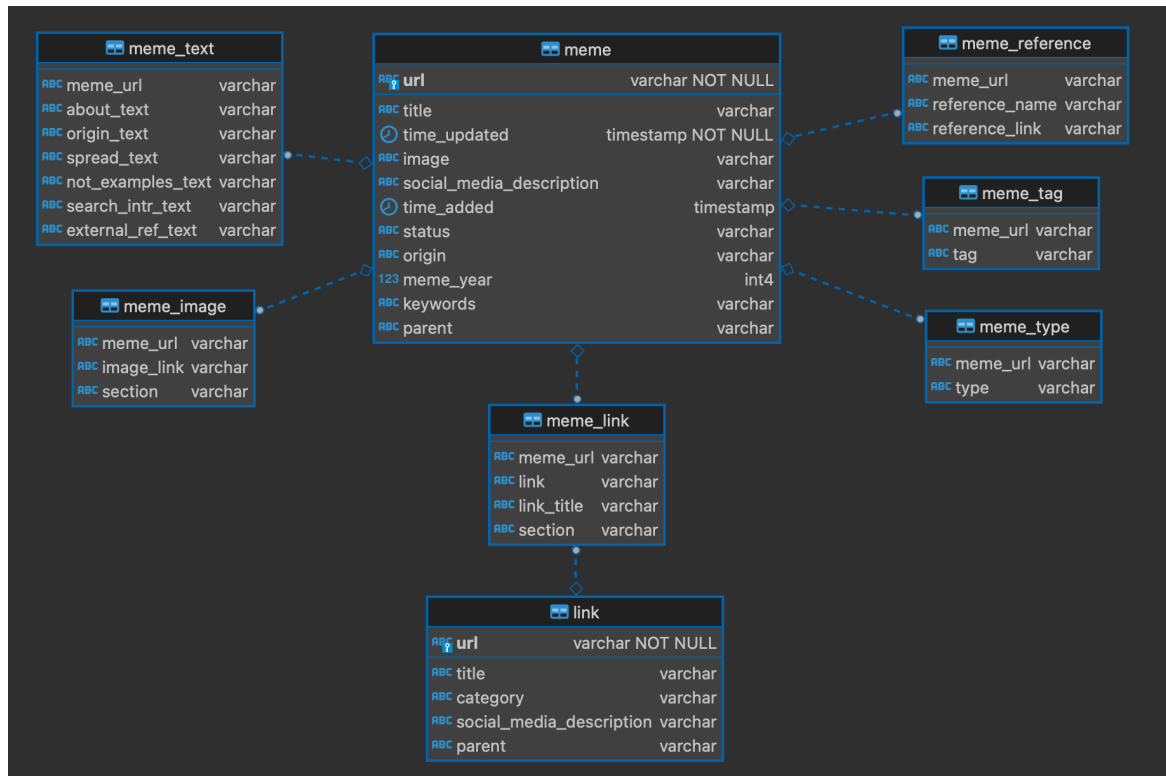
Tables meme\_image, meme\_link, meme\_reference, meme\_tag, meme\_type hold the relationship of meme with image, reference, tag and type. These relationships borrow the key of a meme and should keep the key of another object if it was described in another table, but as we don’t have such data for all of them, only for links. So, the table meme\_tag holds the key of meme (meme\_URL) and the tag. The table meme\_link relates the meme (meme\_URL) and the link (link). It is not wise to hold the key of relationship (if we had one) in the table meme as one meme can have 0 to many references, 0 to many tags, 0 to many types or links or images which makes the meme table enormous.

It turns out that we have the meme and link as dimensions and the relationship tables meme\_image, meme\_link, meme\_type, meme\_tag and meme\_reference as facts describing the relationships between dimensions. As said, we could also have more dimensions: tag, reference, type and image.

Theoretically we could also have tables for statuses and origins. These we could have as dimensions (each meme has only one value for all of these dimensions) and the meme table would be the fact. What would then be other tables. Fact of facts? We see that the dimension

of status would be overthinking as it has only four values, maybe only for descriptive purposes. Table for origin would be good to have and also a meme\_origin relation.

The relational database is accessed with DBeaver. In the figure below you may see the ER diagram of the database in it.



**Figure 2.** ER diagram view in DBeaver.

## 2.3 Graph database

The graph database's first brainstorming session with the whole team yielded the following diagram that can be found [here](#). This session took into account the nodes and properties that we should consider as viable for the project and for future analysis and data mining.

The next step involved further consultation and further brainstorming and refining of the graph database based on its queryability with the following questions being asked:

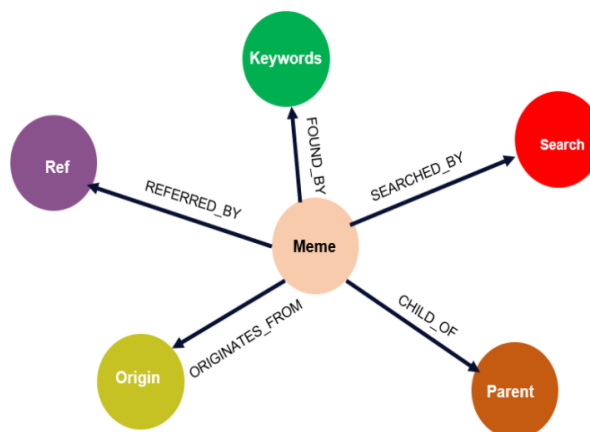
- What are the queries we want to run and their roles?
- What are the current and future goals of the queries in the project?
- How do we want relationships and nodes based on queries?

- How do we want domains and representations to appear in the graph database?
- What is the purpose of each domain?
- How do we want to direct the relationship between the nodes or create the paths?
- What is the role of each node and what properties does it need to capture to achieve the structural aspects of the domain and the relationships (directed-relationships)?

The unwanted categories were then dropped based on the established queries and resulting nodes, their properties and relationships created and defined as follows:

1. **Node-Meme properties:** Title, Image, Tags, SocialMediaDescription, ExtRefText, ExtRefLinks, SpreadText, SpreadImages, SpreadLinks, NotExamplesText, Status, Origin, Type, AboutText, AboutImages, AboutLinks, NotExamplesImages, NotExamplesLinks
2. **Node-Ref properties :** References
3. **Node-Parent properties :** URL
4. **Node-Keywords properties:** Keywords
5. **Node-Origin properties:** OriginText, OriginImages, OriginLinks
6. **Node-Search properties:** SearchIntText, SearchIntImages, SearchIntLinks

A graph database model was then modeled based on selected features as seen in Figure 2.

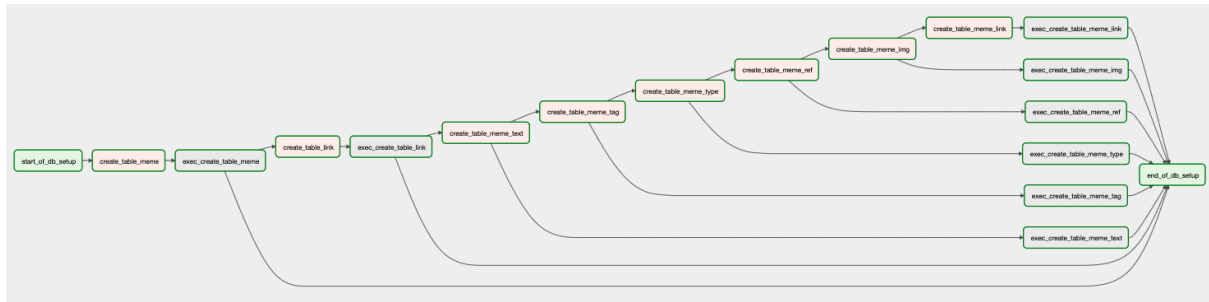


**Figure 3.** Graph database model.

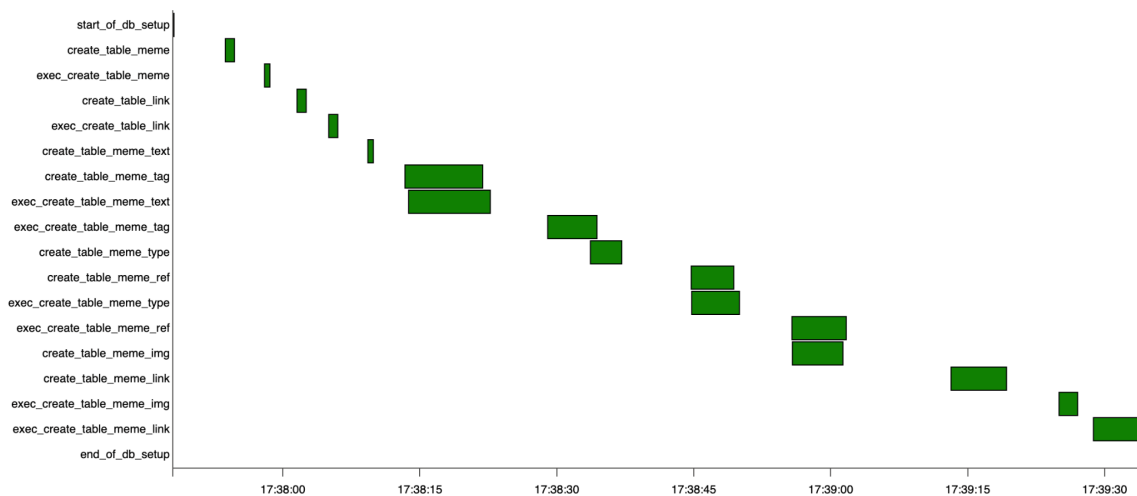
### 3. Pipelines

#### 3.1 Creating tables for relational database

To make the process somewhat easier for the whole team, we decided to make the creation of tables also into a pipeline. This pipeline consists of PythonOperators and PostgresOperators. First ones create the SQL queries for creating the tables and the other one executes them in the PostgreSQL database. The pipeline code is located [here](#) ('pipeline0a.py').



**Figure 4.** View of table creation for the relational database.



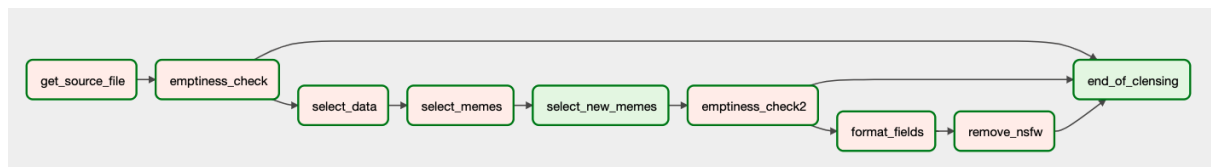
**Figure 5.** Running times for the relational database pipeline.

#### 3.2 Importing data from source to staging area and cleansing

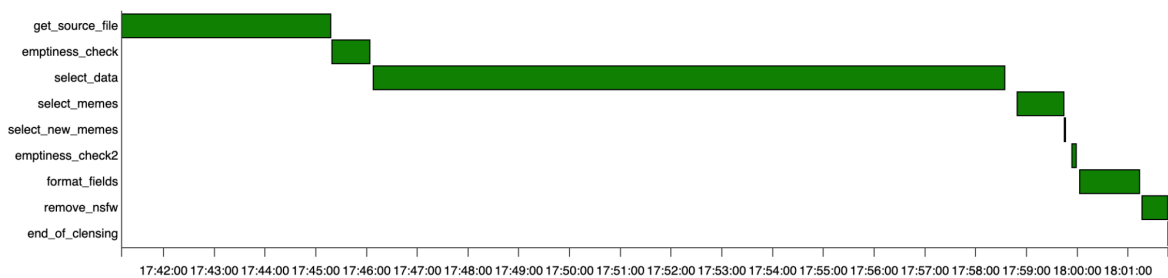
The second pipeline is the longest and most time-consuming, consisting of eight tasks. The first one is a PythonOperator that calls a Python function to download kym.json file into a specified directory and names it after the current epoch to make it unique to the current epoch and easier to find later. The second task is a BranchPythonOperator, it checks if the file is



empty or not. Being a Branch PythonOperator allows the pipeline to continue on to the next task if the file is not empty or go to the end if it is empty. The third task is a PythonOperator, it reads in the downloaded file, flattens it, selects only the wanted features (described in [chapter 2.1](#)) and saves it into a file for the next task. Fourth task reads in the file again, filters out only the memes with category 'meme' and drops duplicate records. If we would have more time for this project, we would make the fifth task check for new rows in the file which makes the pipeline reusable when new data appears in the source. Task six checks if the result file after selection and filtering is empty or not. The seventh task formats epoch time into readable DateTime, cleans strings from unwanted characters and fills in NA/NaN values. The eighth task filters out inappropriate memes by the 'nsfw' tag. The result of the second pipeline is a flattened and cleansed CSV file. The code for the pipeline is available [here](#).



**Figure 6.** First pipeline.

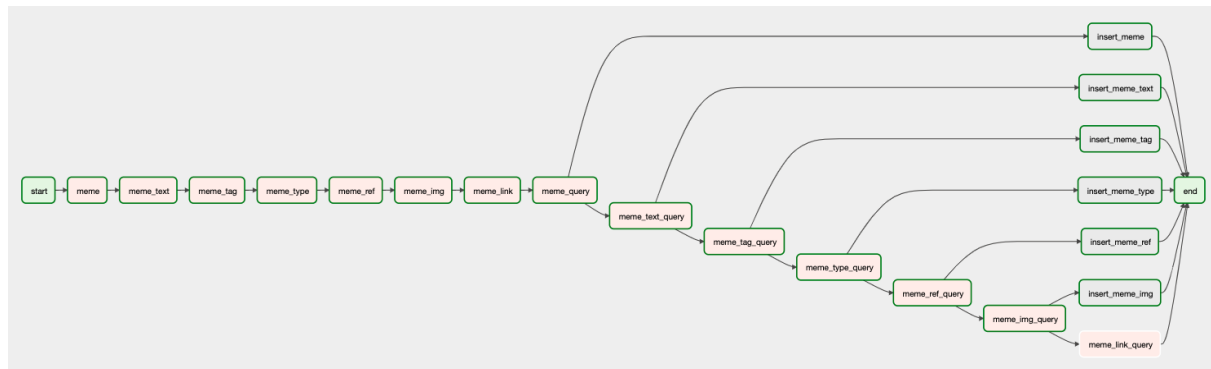


**Figure 7.** Running times for the first pipeline.

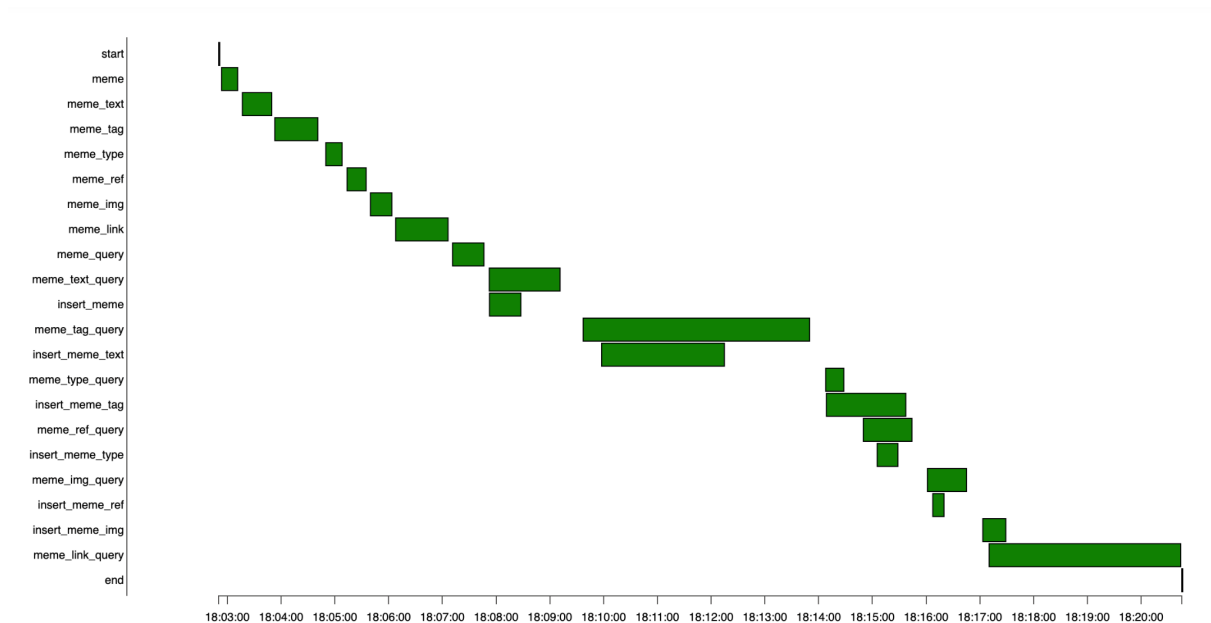
### 3.3 Splitting data for the relational database and importing data to PostgreSQL.

This pipeline consists of PythonOperators and PostgresOperators. PythonOperators read in the CSV file created by the previous pipeline and create SQL queries for the PostgresOperators that execute them in the PostgreSQL database. The data is split into six

tables: meme\_text, meme\_tag, meme\_type, meme\_ref, meme\_img and meme\_link. The table meme\_link is not filled in this pipeline. Current pipeline is visible [here](#).



**Figure 8.** Data splitting pipeline.

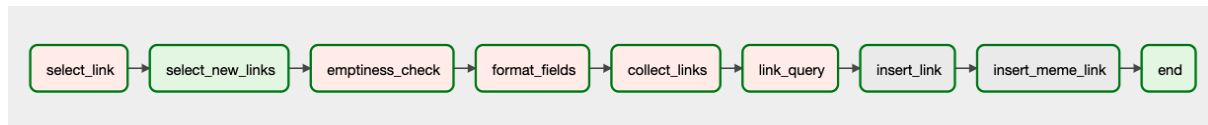


**Figure 9.** Running times for the data splitting pipeline.

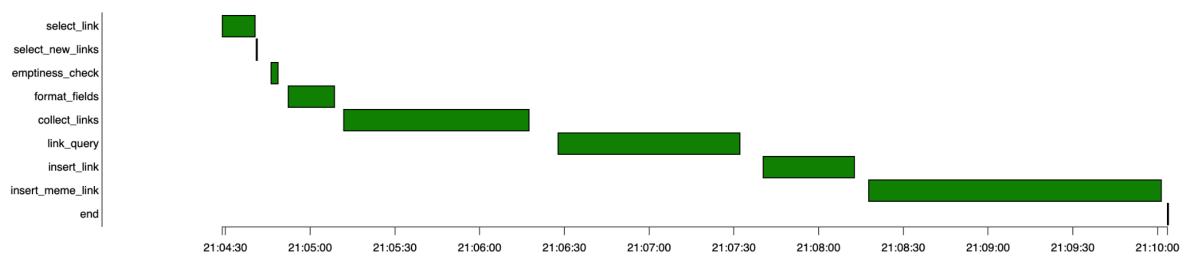
### 3.4 Enriching data using same source

The fourth pipeline uses the file created in the first pipeline, that hasn't yet been filtered by the 'meme' category. It gathers all the links with additional information from the file. If some links are still missing then the information is derived from the link address using natural language processing. Finally the pipeline inserts sourced and derived link data into the table

link. And then it is also possible to insert the meme and link relations to table meme\_link. The pipeline code is [here](#).



**Figure 10.** Data enrichment pipeline.



**Figure 11.** Running times for the data enrichment pipeline.

### 3.5 Importing data to Neo4j

Importation of data in Neo4j involved selecting the best format which everyone agreed to be CSV for our project. As a team we decided not to limit the tools to use but to try learn how to do it using the following tools:

- py2neo and Google colab
- Neo4j desktop
- Airflow

#### Py2neo and Google colab

The First step was to use the cleansed data from [pipeline1](#) to generate a [CSV file](#) for later use with Neo4j desktop. This also gave us an opportunity to use py2neo and Google colab in a collaborative setting where the team could follow in real-time during our usual Friday meetup and later experiment on. The resulting colab file is [indicated here](#). One thing we noted with py2neo was that as much as it's simple to use the documentation needed updating and we wanted the visualization of the graph database and to further explore other neo4j tools.

#### Neo4j Desktop

The next step involved downloading the Neo4j Desktop, installing, doing the basic setup and familiarizing ourselves with the Neo4j Cypher query language that we could use and how we could join it into one query with the import statement as seen [here](#).

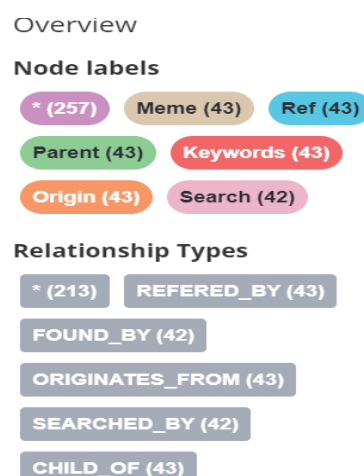
After familiarization with the different Neo4j CQL (cypher query language) commands in the browser the next step was to try to generate a single query that we could later use with Airflow. We started first by creating a new project in Neo4j and adding the graph DBMS and the configuring settings to work with the large meme dataset as [seen here](#).

The final import statement result is indicated in the figure below showing the final single cypher query.

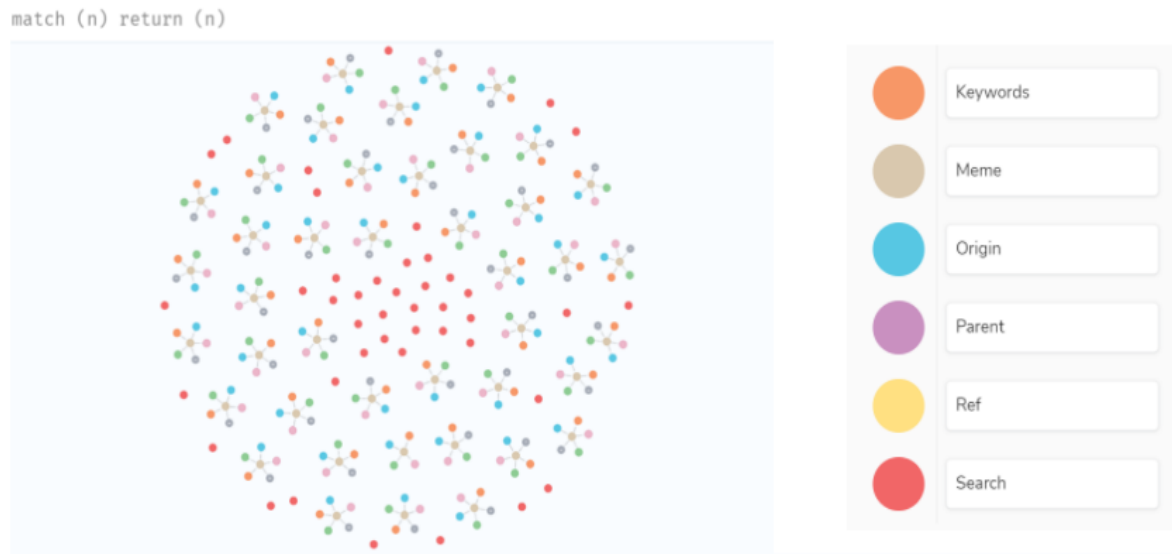


**Figure 12.** Visualisation query.

To visualise the final query all nodes and relationships were called using the following Cypher command **match (n) return (n)** with the overview and relationships indicated below.



**Figure 13.** Created nodes and relationship overview.



**Figure 14.** Visualised graph with nodes and relationships.

## Airflow

Now that the single query generated from Neo4j was ready we were ready to take it to the next level but got disappointed that we could not use the Neo4jOperator due to import of `airflow.providers.neo4j.operators` error.

## 4. Analysis

### 4.1 Validation of relational database

In order to check and test the quality of the data loaded we made some simple validation queries for each loaded table. We counted the rows, distinctive values, empty values for different features, took simple counts over groups, etc. Validation queries can be found [here](#).

### 4.2 Analysis in the relational database

The analysis of the relational database was made with DBeaver. Some of the analysis can be seen in the next chapters. SQL file with analyse queries is saved [here](#).

#### 4.2.1 Parents, children, grandchildren, ...

Through several joins and executions after each join we could find out all the first parents, the parents who do not have parents, their children, grandchildren, etc. We got 7 generations of memes. You can see the upper part of the table below.

	ABC parent0	ABC parent1	ABC parent2	ABC parent3	ABC parent4	ABC parent5	ABC parent6
1	Memes	Ironiic Memes	Smiling Man / Wrongly Named Memes	Spurdo Spärde	American Bear	Freedom Aint Free	[NULL]
2	Memes	Image Macros	Advice Animals	Scumbag Steve	Scumbag Hat	Scumbag America	[NULL]
3	Memes	Image Macros	Advice Animals	Scumbag Steve	Scumbag Hat	Scumbag Brain	[NULL]
4	Memes	Image Macros	Advice Animals	Scumbag Steve	Scumbag Hat	Scumbag Christian	[NULL]
5	Memes	Image Macros	Advice Animals	Scumbag Steve	Scumbag Hat	Scumbag DNA	[NULL]
6	Memes	Image Macros	Interior Monologue Captioning	Captioned Stock	Are You Sexually	Triple the Dosage	[NULL]
7	Memes	Copyasta	Creepypasta	Trollpasta	And Then A Skel	[NULL]	[NULL]
8	Memes	Image Macros	Advice Animals	Socially Awkward	Awkward Momei	[NULL]	[NULL]
9	Memes	Image Macros	Advice Animals	Courage Wolf	Baby Courage W	[NULL]	[NULL]
10	Memes	Image Macros	Advice Animals	Insanity Wolf	Baby Insanity W	[NULL]	[NULL]
11	Memes	Image Macros	Advice Animals	Advice Dog	Bad Advice Cat	[NULL]	[NULL]
12	Memes	Image Macros	Interior Monologue Captioning	Captioned Stock	Because Your M	[NULL]	[NULL]
13	Memes	Ironiic Memes	Smiling Man / Wrongly Named Memes	Spurdo Spärde	Benis	[NULL]	[NULL]
14	Memes	Image Macros	Interior Monologue Captioning	Captioned Stock	Bill and Phil / I H	[NULL]	[NULL]
15	Memes	Image Macros	Advice Animals	Confession Bear	Confession Kid	[NULL]	[NULL]
16	Memes	Ironiic Memes	Smiling Man / Wrongly Named Memes	Spurdo Spärde	Ebin	[NULL]	[NULL]

Figure 15. Screenshot of 7 generations of memes.

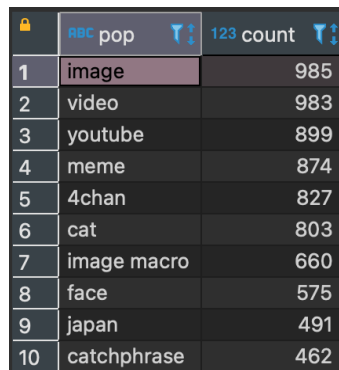
This outcome was derived with the following query.

```
— Relatives ordered by the youngest generations existing
select * from (
  select distinct
    m1.title as parent0,
    m2.title as parent1,
    m3.title as parent2,
    m4.title as parent3,
    m5.title as parent4,
    m6.title as parent5,
    m7.title as parent6
  from meme m1
  left join meme m2 on m2.parent = m1.url
  left join meme m3 on m3.parent = m2.url
  left join meme m4 on m4.parent = m3.url
  left join meme m5 on m5.parent = m4.url
  left join meme m6 on m6.parent = m5.url
  left join meme m7 on m7.parent = m6.url
  where m1.url in (select distinct m.parent from meme m where m.parent != '')
  and m1.parent = ''
) relatives
order by 7,6,5,4,3,2,1 desc;
```

Figure 16. Screenshot of meme parents query.

### 4.2.2 Most popular tags and tags related

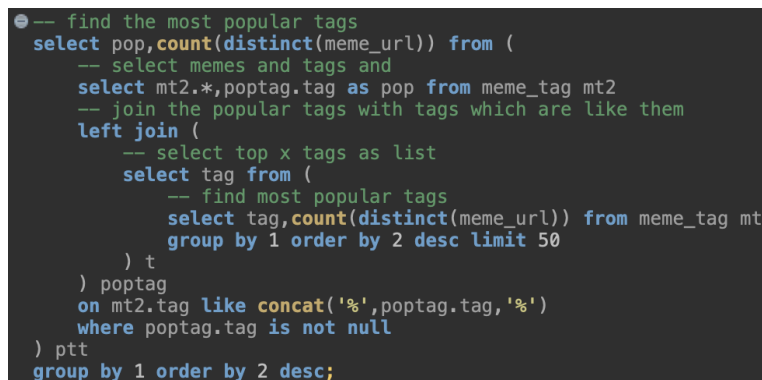
Finding the most popular tags was not a straightforward finding. First, we found 50 most frequent tags. Then we join them with tags which were like these top 50 tags. We took new counts of the popular tags again and got the final top. In the following figure you can see the top10 of these.



	asc pop	123 count
1	image	985
2	video	983
3	youtube	899
4	meme	874
5	4chan	827
6	cat	803
7	image macro	660
8	face	575
9	japan	491
10	catchphrase	462

**Figure 17.** Screenshot of top 10 tags.

This result is derived with the following query.



```
-- find the most popular tags
select pop,count(distinct(meme_url)) from (
  -- select memes and tags and
  select mt2.*,poptag.tag as pop from meme_tag mt2
  -- join the popular tags with tags which are like them
  left join (
    -- select top x tags as list
    select tag from (
      -- find most popular tags
      select tag,count(distinct(meme_url)) from meme_tag mt
      group by 1 order by 2 desc limit 50
    ) t
  ) poptag
  on mt2.tag like concat('%',poptag.tag,'%')
  where poptag.tag is not null
) ptt
group by 1 order by 2 desc;
```

**Figure 18.** Screenshot of the top 10 tags query.

### 4.2.3 Timeline of the added memes by origins

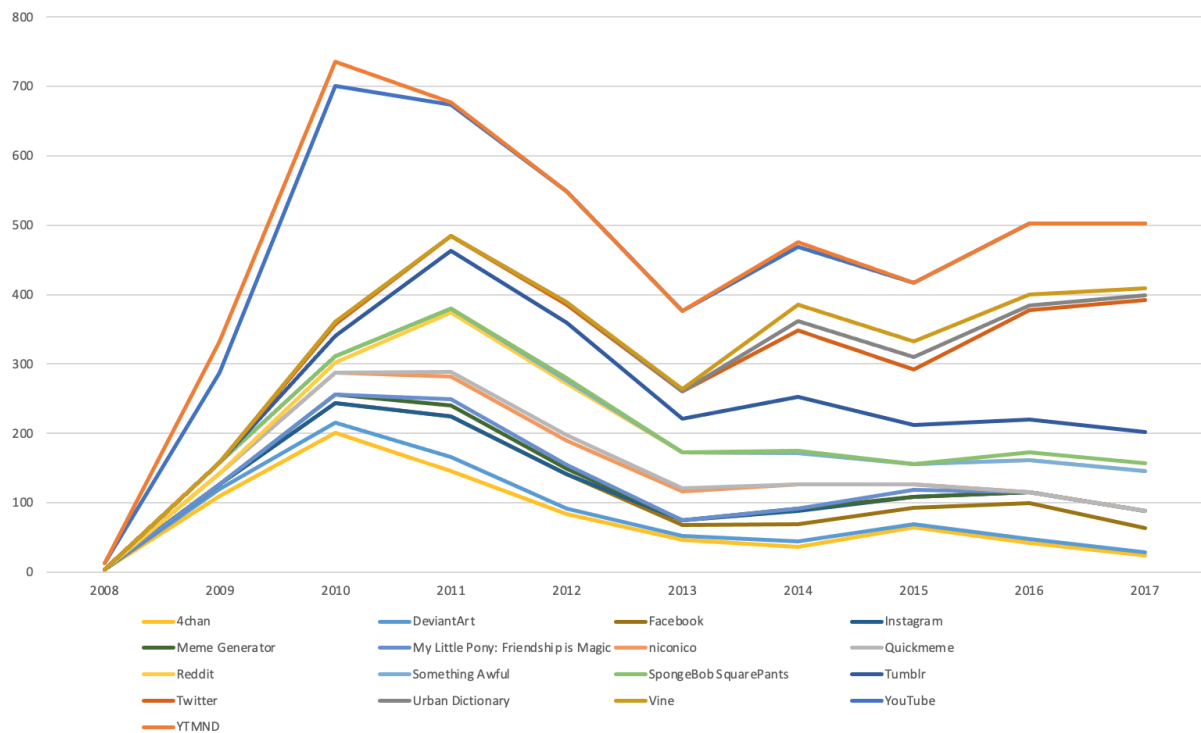
One of the weaknesses of relational databases and SQL is the part of analysis when we want to get any illustrations or plots and graphs, we need to use other applications. In the following analysis we used the ordinary MS Excel.

We wanted to know the timeline and counts for different origins. Query was very simple.

```
-- time added year and origin counts
select extract(year from time_added), origin, count(distinct(m.url)) from meme m
group by 1,2 order by 3 desc;
```

**Figure 19.** Screenshot of meme origins timeline query.

In the next step the table was copied to MS Excel and turned into a pivot table. We filtered out only origins with bigger counts of memes and then the plots were generated.



**Figure 20.** Timeline of meme origins.

#### 4.2.4 Analysing links

Relationships between memes are one of the interesting topics in the dataset. After analysing the children and parents, we also discovered that some of the links were memes. So we counted how many of them. In the figure below is the statistics of it.



	ABC section	ABC is_meme	123 count
1	External Reference	N	56,184
2	Spread	N	3,215
3	Origin	N	2,363
4	About	N	2,250
5	Spread	Y	1,295
6	About	Y	1,152
7	Origin	Y	698
8	Notable Examples	N	233
9	Search Interest	N	101
10	Notable Examples	Y	62
11	External Reference	Y	16
12	Search Interest	Y	12

**Figure 21.** Screenshot of the count of links that were also memes.

This table was a result of the following query.

```

select
  section,
  case
    when m.url is null then 'N'
    when m.url is not null then 'Y'
  end as is_meme,
  count(distinct(link))
from meme_link ml
left join meme m
on m.url = ml.link
group by 1,2 order by 3 desc;

```

**Figure 22.** Screenshot of the query to find links that are also memes.

### 4.3 Analysis in the graph database

For fast graph database lookup of nodes and relationships we chose indexes over constraints since generated nodes lacked id's and duplicates had already been dropped in the data cleansing stage.

#### Time analysis

The three randomly selected queries from feature selection and graph modeling that were discussed and then used to find the speed and total time taken to run each query.

1. Status of memes created and their count.
2. Finding memes based on their origin.
3. Defining the source of memes and total count in each source.

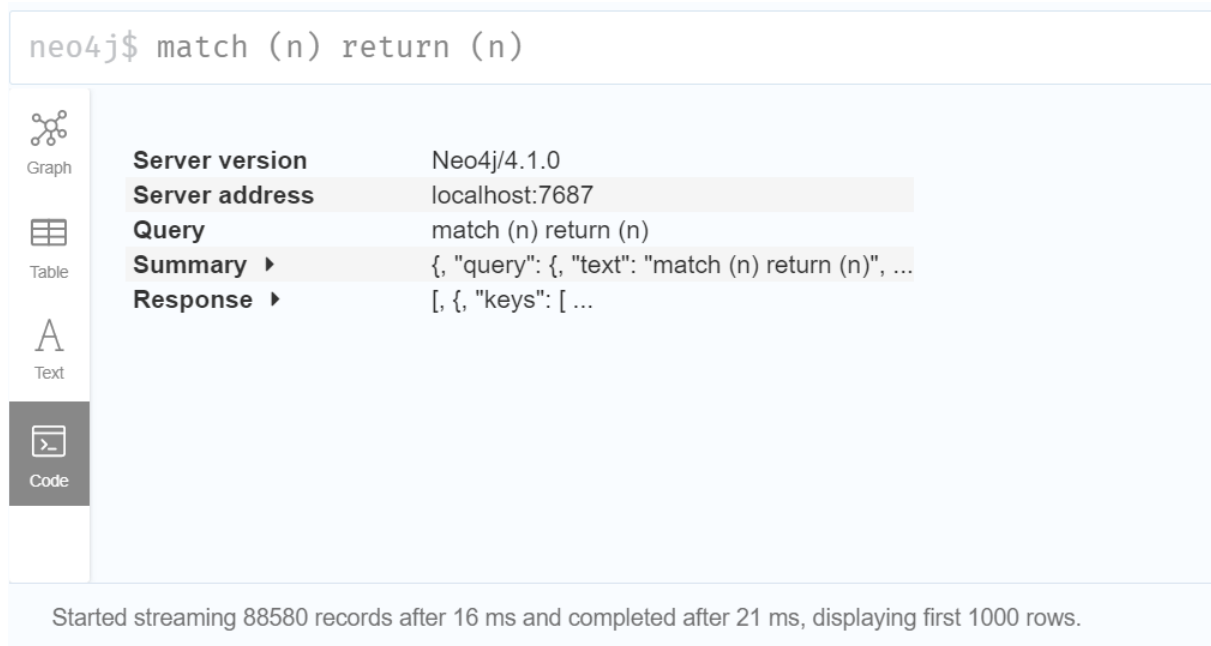
The image results can be found [here](#).

Query	Results	Speed (Total time taken)
<b>MATCH</b> (meme:Meme) <b>RETURN</b> meme.status <b>AS</b> STATUS, <b>count</b> (meme) <b>AS</b> Count <b>ORDER BY</b> <b>count</b> (meme) <b>DESC</b> <b>LIMIT 300</b>	Started streaming 4 records in less than 1 ms and completed after 16 ms.	4 records/<1ms (16ms)
<b>MATCH</b> (origin:Origin)<-[:ORIGINATES_FROM]-(meme:Meme) <b>WHERE</b> origin.originLinks= '[[MTVMusic Awards, <a href="https://knowyourmeme.com/memes/events/mtv-video-music-awards">https://knowyourmeme.com/memes/events/mtv-video-music-awards</a> ]]' <b>RETURN</b> meme.title <b>AS</b> Title, <b>count</b> (meme.title) <b>AS</b> Count <b>ORDER BY</b> <b>count</b> (meme.title) <b>DESC</b>	Started streaming 1 records after 15 ms and completed after 15 ms.	1 record/>15ms (15ms)
<b>MATCH</b> (meme:Meme) <b>RETURN</b> meme.origin <b>AS</b> Source, <b>count</b> (meme) <b>AS</b> Count <b>ORDER BY</b> <b>count</b> (meme) <b>DESC</b> <b>LIMIT 300</b>	Started streaming 300 records in less than 1 ms and completed after 30 ms	300 records/30ms (30ms)

**Table 1.** Table showing the performance of queries in relation to time.

The speed indicated that the time taken to traverse the data structure occurred at a high speed indicating the fast lookup between nodes and given relationships.

Matching of all created nodes and their relationships takes 21ms for 88580 records.



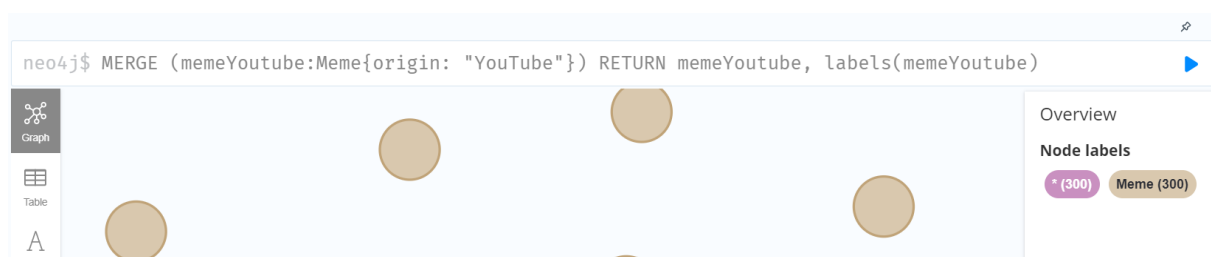
**Figure 23.** Time taken to visualise all nodes and their relationship and code overview.

## Flexibility

The flexibility of creating new relationships and nodes without affecting existing queries can be seen for example when use commands like merge to forge new memes as seen using:

**MERGE** (memeYoutube:Meme{origin: "YouTube"})

**RETURN** memeYoutube, labels(memeYoutube)



**Figure 24.** Query merge for all memes with origin being Youtube.

## Ease in

- Use-Learning curve: The Neo4j CQL was easy for the team once we understood what needed to be changed in settings of the graph RDMS. Lack of Joins was also a welcomed change.
- Data retrieval and manipulation.
- Representing connected and semi-structured data.

**Main weakness noted with Neo4j was:**

- Data storage had to change the memory settings in order to run the single query or import the CSV without getting an error. The other option was to limit query based on availability of space in the memory settings.
- Storage space increments was a main indicator of problems with scalability that may occur in a project when it comes to cost incurred with storage use.

The conclusion was that it's most suited for big datasets that are connected, use complex join queries and need high performance in terms of speed as with live data like filtering bad or sensitive memes as fast as possible. Our project dataset was not as highly connected as we would have wanted so it felt that in the end the nodes and relations would have been designed with more connections to increase complexity of the joins in order to maximise the use of the graph database.

## 5. Teamwork

### 5.1 Initial plan

The initial step was for everybody to familiarise themselves with the raw json file. Then we had a few brainstorming sessions for a vision of the database and imagining tasks that we want to do with the data. Next we came up with a relational monutable schema which we intended to use as the final relational database model, as we did not have queries that would have benefited from splitting up the database.

We divided the tasks of the first planned pipeline between our team members. We already then saw that some tasks were bigger than others but that was the starting phase. Then we realised that the next tasks were impossible to create before the previous were ready. So, naturally we formed pairs to help out each other.

We decided to use Neo4j for the platform for our graph database model and implementation as we had some experience with it from the practicals of this course. As we did not have the relational database ready we decided to split our work in two pairs. The first pipeline was left to Kertu and Heidi to finish it. Vera and Mart dived into a whole new topic creating the graph database. Now the work of pairs was only under general discussions in the whole team.

### 5.2 Final outcome

The final relational database model consists of eight tables describing the main object 'meme' and additional object 'link' as dimensions and all relationships in fact tables. This design for this schema and implementation was by Heidi and Kertu. The Neo4j graph database design and implementation was by Vera and Mart. Databases are created and populated by five pipelines.

This is the result of our teamwork which was very flexible. Everybody reacted quickly to needed changes, invested a lot of effort and helped each other when needed. It is important to use the strengths of the members and share the knowledge as quickly as possible.

## 6. Lessons learned

### 6.1 Challenges constructing pipelines

Working with AirFlow was sort of a challenge due to the documentation not being the greatest and AirFlow itself not running that well, which could also just be a problem with our computers. But the need to rerun time-consuming pipelines due to unrelated errors was not always fun and took unnecessarily much time. Fortunately it was possible to not have to run the whole pipeline every time, for example to not download and cleanse the file every time but just use one already created during a previous run.

### 6.2 Challenges building relational db

As described before the main issue with building the relational database was to find out the data split into dimensions and facts. The memes topic is not a usual business area and the mechanical split into facts and dimensions did not give the expected result. If we would have the meme as a fact we would have a large number of meme records to hold. So, we decided to solve our relational database vice versa.

### 6.3 Challenges creating graph db

- Graph DB designed how to implement the nodes and their relationships. In the end towards the end of the project realised that we needed to create nodes with complex relationships.
- Initial setup of Neo4j desktop to read csv with large dataset and experiment with the dataset. While working as a pair we did not realise that most of the settings were commented out so we needed to uncomment them in order to load the CSV file without setting limits.
- Finding the updated queries for py2neo since the shared one in course repository some commands no longer apply and also dealing with the neo4jOperator in Airflow throws an error even after following documentation and example given on how to use it.
- Combining the final Cypher queries to use for the pipeline in Airflow with Neo4j but on more reading and exploring it was clear on how to proceed.
- How one can combine CREATE INDEX FOR and CREATE in one query yet to figure that out.

## **Summary**

Some aspects of the project were extremely difficult and time consuming, but we learned a lot. The design of the pipelines was influenced mostly by the design of relational databases as our team had more experience with relational databases rather than graph databases.

Our teamwork was more than satisfying. It was enjoyable. We were flexible and tightly connected. We discussed a lot and got to know each member in the team even without seeing each other at all. We shared our issues, asked for help, reacted quickly and supported each other.

**Extra 1.** Schema of the relational database in PostgreSQL.

