# Data Engineering Project: Analyzing Scientific Publications

Fall 2023/2024 LTAT.02.007 University of Tartu

==================================

**Team 17: Erkki Tikk, Mattias Väli, Kristjan Laht, Anton Malkovski**
Public repository: https://github.com/etikk/data-engineering-project

This project is part of a Data Engineering course, focusing on the design and implementation of a data pipeline to analyze scientific publications data. The primary objectives include building a multi-faceted analytical view on scientific publications data, creating a data warehouse/data mart for Business Intelligence (BI) queries, utilizing a graph database for co-authorship prediction and community analysis, and implementing various data transformations, cleansing, and augmentation processes.

The technologies and systems used in this project encompass Python as the programming language, PostgreSQL/MySQL for the data warehouse, Neo4J for the graph database, Apache Airflow for data pipeline management, pgAdmin for PostgreSQL management, and Docker and Docker Compose for containerization. The project also utilizes Python libraries like Pandas, Requests, SQLAlchemy, and Py2neo, along with data transformation tools such as scholar.py, Crossref API, etc.

The project setup involves Docker and Docker Compose installation, cloning the project repository, obtaining a .env file with the necessary environment variables, building and running Docker containers, setting up the database in pgAdmin, accessing Neo4J Browser, and setting up AirFlow.

Let's break down the configuration:

## Services:

db (PostgreSQL):

- Image: PostgreSQL official image.
- Exposes the PostgreSQL database on port 5432.
- Defines environment variables for database configuration.
- Uses a volume (postgres_data) to persist data.

pgadmin:

- Image: pgAdmin 4 image.
- Exposes pgAdmin on port 5050.
- Depends on the db service.

airflow-webserver:

- Builds an image from the current directory (.).
- Sets up an Airflow web server.
- Configures environment variables for connecting to the PostgreSQL database.
- Exposes Airflow web server on port 8080.
- Depends on the db service.
- Mounts volumes for Airflow configuration, DAGs, logs, plugins, and raw data.

airflow-scheduler:

- Similar to airflow-webserver but for the Airflow scheduler.

neo4j:

- Image: Neo4j official image.
- Exposes Neo4j on ports 7474 (HTTP) and 7687 (Bolt).
- Defines environment variables for Neo4j authentication.
- Uses volumes for persisting Neo4j data, imports, logs, and plugins.

## Volumes:

- postgres_data: Volume for persisting PostgreSQL data.
- neo4j_data: Volume for persisting Neo4j data.
- neo4j_import: Volume for Neo4j import data.
- neo4j_logs: Volume for Neo4j logs.
- neo4j_plugins: Volume for Neo4j plugins.

## Environment Variables:

The configuration file uses environment variables for sensitive information such as database passwords, Neo4j credentials, and Airflow configuration.

## Dependencies:

- The pgadmin service depends on the db service.
- The airflow-webserver and airflow-scheduler services depend on the db service.

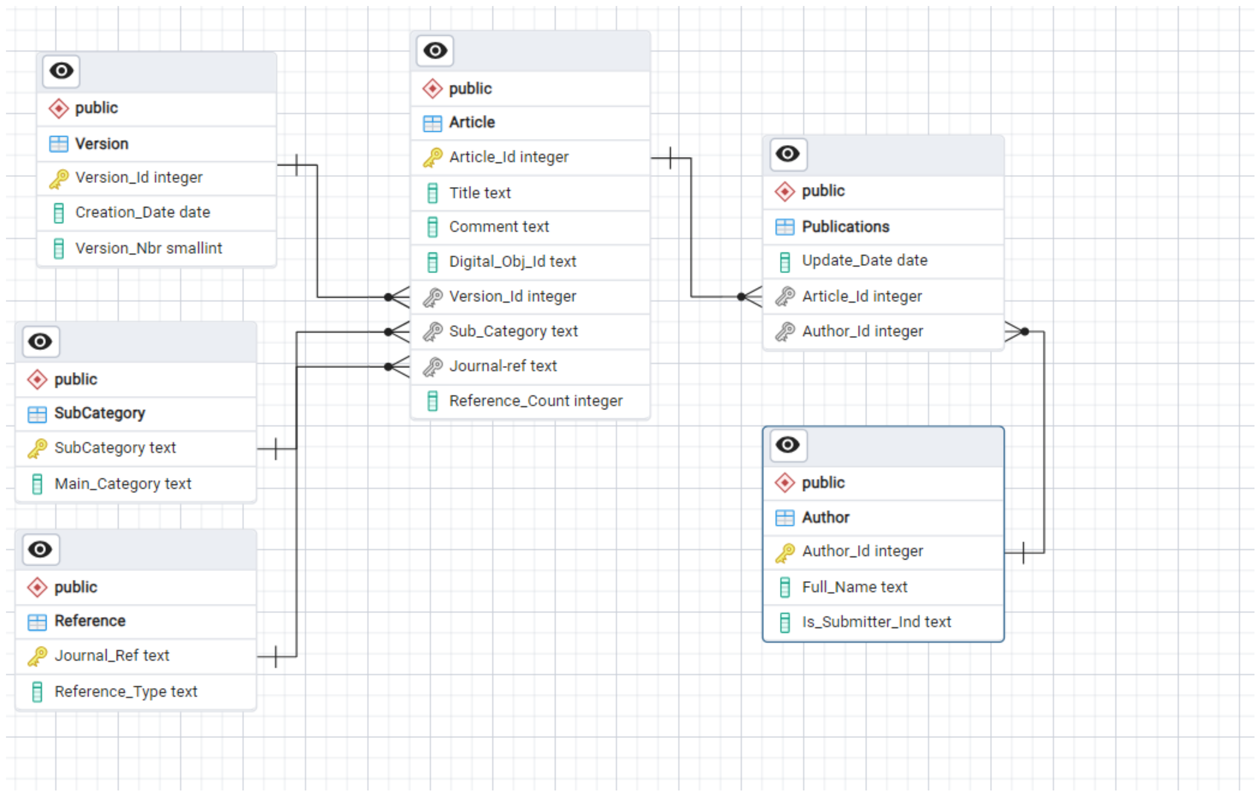
## Notes:

- The build: . directive in airflow-webserver and airflow-scheduler indicates that Docker should build an image from the current directory, suggesting there's a Dockerfile in the same directory for these services.

- The command directive specifies the command to run when starting the containers.
- Some commands in the comments (#command: webserver and #command: scheduler) suggest that you might uncomment and use these as alternative ways to start the containers.

Overall, this Docker Compose configuration sets up a development environment with PostgreSQL, pgAdmin, Apache Airflow, and Neo4j, all orchestrated using Docker Compose.

## Database schema:



This database has six different tables: Version, Article, Category, Publications, Author, and Reference. Here's a brief description of each:

- **Version**: has attributes like Version_ID (integer), Creation_Date (date), and Version_Nbr (smallint).

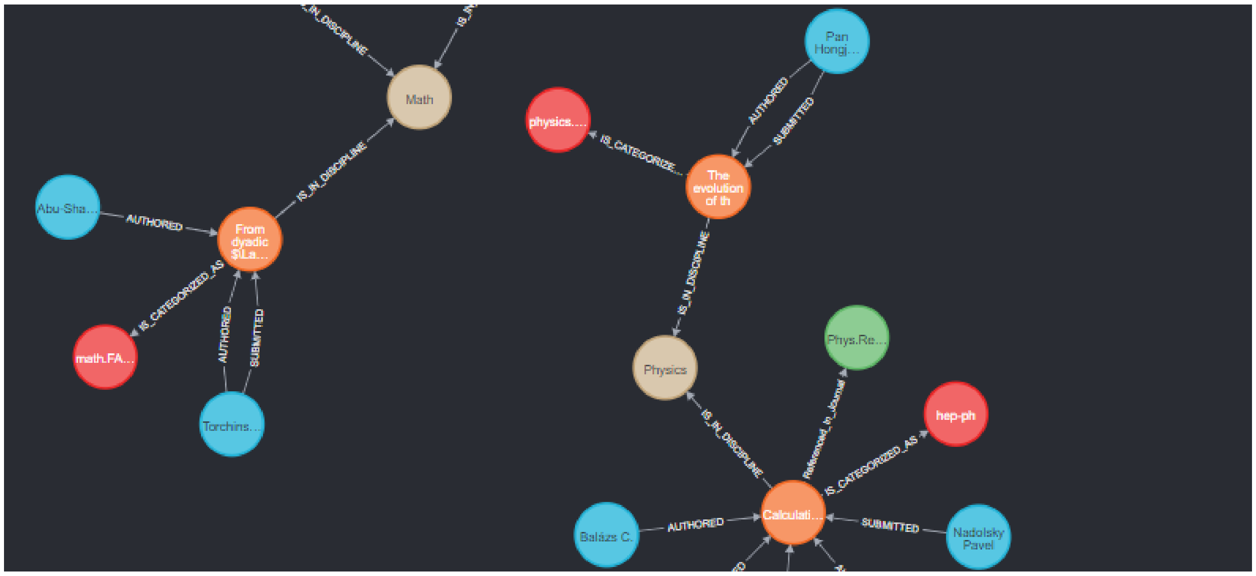
- **Article**: includes Article_ID (integer), Title (text), Comment (text), Digital_Obj_Id (integer), Category_ID (integer), and Abstract (text).
- **Category**: consists of Category_ID (integer) and Category_Desc (date).
- **Publications**: is connected to the "Article", containing Article_ID(integer), Author_ID(integer), and Reference_ID(integer).
- **Author**: table has Author_ID(integer), Full_Name(text), and Is_Submitter(text).
- **Reference**: table includes Reference_ID(integer) and Referencing_Article_Id(integer).

The "Article" table is linked to "Version", "Category", "Publications" while "Publications" is further connected to "Author". This schema represents the structure of a database and how data is organized within it.

This database schema can support a variety of Business Intelligence (BI) queries. Here are a few examples:

1. Article Analysis: You can retrieve all articles within a specific category or written by a specific author. This can help in understanding the distribution of articles across different categories or authors.
2. Author Contribution: You can find out which authors have submitted the most articles, which can be useful for recognizing prolific contributors.
3. Version Tracking: You can track the versions of a particular article to see its evolution over time.
4. Reference Analysis: You can analyze the references made by articles to understand the interconnection between different articles.

## GRAPH VIEW



Here is a brief explanation of the relationships in Neo4j database.

Authored Relationship:

- Definition: MERGE (a:Author)-[:Authored]->(b:Title)
- Description: Represents the relationship between an author (node labeled "Author") and a title (node labeled "Title"). This relationship indicates that the author has authored the specified title.

Submitted Relationship:

- Definition: MERGE (c:Author)-[:SUBMITTED]->(b:Title)
- Description: Represents the relationship between a submitter (node labeled "Author") and a title (node labeled "Title"). This relationship indicates that the submitter has submitted the specified title.

Is_Categorized_As Relationship:

- Definition: MERGE (b:Title)-[:IS_CATEGORIZED_AS]->(d:Categories)
- Description: Represents the relationship between a title (node labeled "Title") and a categories node (node labeled "Categories"). This relationship indicates that the title is categorized under the specified category or categories.

Referenced_In_Journal Relationship:

- Definition: MERGE (b:Title)-[:Referenced_In_Journal]->(e:Journal_Reference)
- Description: Represents the relationship between a title (node labeled "Title") and a journal reference node (node labeled "Journal_Reference"). This relationship indicates that the title is referenced in the specified journal.

These relationships collectively form a graph that represents the connections between authors, titles, submitters, categories, and journal references. Each relationship type signifies a different aspect of the data model and provides insights into how entities in the graph are related to each other. The graph structure allows for querying and traversing relationships to gain a comprehensive view of the data and its associations.

## Data transformation and enrichment

Our project, integrated within Apache Airflow, orchestrates a comprehensive DAG pipeline designed for enhancing research paper datasets with extensive citation data and category-based insights. The DAG is structured as follows:

Initial Data Processing: The pipeline initiates by processing a collection of JSON files stored in /app/raw_data. These files, each named with a 'chunk' prefix, are systematically filtered through the filter_publications task. This step focuses on extracting research papers based on specific categories and titles, discarding non-essential data like abstracts for efficiency.

Category Augmentation: Next, the augment_with_categories task takes over, utilizing a CSV file (_Cat_Csv.csv) and a JSON file (Citations_Pubtype.json) to enrich each research paper with broader category data. This process involves a series of data frame manipulations, where each paper's title is mapped to relevant categories and descriptions, significantly enhancing the dataset's categorical depth.

Citation Count Enrichment: The augment_with_citations task further enriches the dataset by querying the Crossref API for Digital Object Identifiers (DOIs) and citation counts based on the titles. Each paper's citation count is meticulously compiled, providing a valuable metric of its academic impact.

Database Integration: The enriched data is then transitioned into two distinct database systems via save_to_postgres and save_to_neo4j tasks. PostgreSQL is utilized for structured data storage, while Neo4j serves as a graph database to capture the complex relationships within the dataset. These databases are meticulously structured with various tables and indices to optimize query performance and data retrieval.

Automated and Periodic Execution: The DAG is configured for periodic execution, ensuring that the dataset remains up-to-date with the latest citation counts and category information. This automated process allows researchers and data analysts to continuously access enriched data without manual intervention.

Scalability and Customization: The pipeline's modular design, reflected in distinct PythonOperator tasks within the DAG, allows for easy scalability and customization. Researchers and data practitioners can modify or extend this pipeline to suit specific data processing needs or to integrate additional data sources.

## BI Queries

This type of data analysis is a form of Business Intelligence (BI) in the academic context, aiding scholars in their research. In broad terms, these queries facilitate the discovery and evaluation of academic articles and authors. By identifying duplicate authors, academics can ensure the accuracy of their bibliographic references. Querying for authors with similar names aids in disambiguating authors with common names, improving the precision of author-based searches.

Queries that rank authors by the number of publications or calculate h-indexes allow users to identify the most prolific or impactful authors in a field. This can guide researchers in finding high-quality articles or potential collaborators. The ability to fetch articles by specific authors in JSON format can provide a convenient way for users to integrate this data into other software tools or workflows.

The specific queries are provided in the Appendix, but here follows a short description for each SQL query (or query category):

1. Identifying Duplicate Authors: These queries are useful for data cleaning and integrity checks. They help to identify if there are duplicate entries in the `author` table, which might occur due to data entry errors or inconsistencies in naming conventions. The first query counts the total number of authors, and the next three queries are variations of finding duplicate authors by grouping by the `full_name` field.
2. Searching for Specific Authors: This query retrieves authors with names similar to 'Steeghs'. It uses the `LIKE` operator, which can be useful when you want to search for a pattern in a text field, such as finding all authors with names starting with a specific string.
3. Ranking Authors by Publications: This query ranks authors based on the number of publications they have. It could be used to identify the most prolific authors. It uses a subquery to count the number of publications for each author, and then joins this with the `author` table to get the author names.
4. Calculating H-Index: This query calculates the h-index for each author, which is a common measure of an author's productivity and impact in academia. It uses the window function `RANK()` to rank the articles by citation count for each author, and then calculates the h-index as the maximum rank where the rank is less than or equal to the citation count.
5. Fetching Articles by Specific Authors in JSON Format: This is a complex query that fetches articles written by authors of a specific article (in this case, where `article_id` is 10) and returns the result in JSON format. It uses the `json_agg` and `row_to_json` functions to format the query result as JSON. This could be useful when the query result needs to be returned as JSON, for example, to be consumed by a web application or API.

Here follows a short description for Cypher queries:

1. The first query finds authors who frequently work on the same titles or categories, useful for identifying research groups or themes.
2. The second query finds authors who frequently co-author papers, useful for identifying strong collaborations within the network.
3. The third query applies the PageRank algorithm to rank authors by influence, useful for identifying key influencers within the community.

In the context of BI, these queries allow academics to make data-driven decisions about which articles to read, which authors to follow, and which topics are gaining traction in their field. This can lead to more efficient use of their time and potentially uncover new directions for their own research,

## Conclusion

The project successfully demonstrated the potential of leveraging data engineering and business intelligence techniques in the analysis of scientific publications data. The combination of Python, PostgreSQL/MySQL, Neo4J, Apache Airflow, pgAdmin, Docker, and various Python libraries enabled the creation of an effective data pipeline and data warehouse. This setup facilitated the enrichment of research paper datasets and the execution of insightful BI queries. The implemented system can support academics in their research by enabling data-driven decisions, improving the accuracy of bibliographic references, and identifying influential authors and potential collaborators. Future work could further optimize the data transformation and enrichment process, expand the range of BI queries, and improve the system's scalability to handle larger datasets.

## Individual Contribution statement

Erkki Tikk - repo, Docker, PostGres, Neo4j, pgAdmin, AirFlow setup and integration, PostGres task, Filtering task, AirFlow pipeline DAG

Mattias Väli - PostGres and Neo4j DB schemas, Neo4j task, Citations API integration, Categories augmentation

Anton Malkovski - Project Design Document

Kristjan Laht - BI queries, pipeline reordering, logic fixes, db optimization, data-loss fixes, data enrichment bug fixes and optimization

# Appendix

## SQL queries

```sql
-----------------------------------------------------
-- queries to check if we have many duplicate authors
-- if the results between the different queries here differ
-- then something is wrong
SELECT count(*)
FROM author;

SELECT count(*), full_name
FROM author
GROUP BY (full_name)
ORDER BY count(*) DESC
LIMIT 100;

SELECT sum(grp)
FROM (SELECT 1 AS grp
     FROM author
     GROUP BY (full_name)) AS t;
-----------------------------------------------------


-- see how many similarly named authors there are to Steeghs
SELECT *
FROM author
WHERE full_name LIKE 'Steeghs%';

-- rank authors by number of publications
SELECT counts, full_name
FROM (SELECT count(author_id) AS counts, author_id
     FROM publications
     GROUP BY (author_id)
     ORDER BY count(author_id) DESC
     LIMIT 100) AS t
        LEFT JOIN author ON author.author_id = t.author_id;


-- select authors by h-index
SELECT author_id, MAX(rank) AS h_index
FROM (SELECT p.author_id,
            a.article_id,
            a.cited_by_count,
            RANK() OVER (PARTITION BY p.author_id ORDER BY a.cited_by_count
DESC) AS rank
     FROM publications p
```

```
               JOIN article a ON p.article_id = a.article_id) subquery
WHERE rank <= cited_by_count
GROUP BY author_id
ORDER BY h_index DESC;


-- get articles by the same authors where article_id == 10. in JSON !
SELECT coalesce(json_agg("root"), '[]') AS "root"
FROM (SELECT row_to_json(
                    (SELECT "_e"
                     FROM (SELECT "_root.ar.root.publications"."publications"
AS "publications",
                                  "_root.base"."title"
AS "title",
                                  "_root.base"."abstract"
AS "abstract") AS "_e")
            ) AS "root"
     FROM (SELECT *
           FROM "public"."article"
           WHERE (
                     ("public"."article"."article_id") = (('10') :: integer)
                     )) AS "_root.base"
              LEFT OUTER JOIN LATERAL (
         SELECT coalesce(json_agg("publications"), '[]') AS "publications"
         FROM (SELECT row_to_json(
                              (SELECT "_e"
                               FROM (SELECT
"_root.ar.root.publications.or.author"."author" AS "author") AS "_e")
                      ) AS "publications"
               FROM (SELECT *
                     FROM "public"."publications"
                     WHERE (("_root.base"."article_id") = ("article_id"))) AS
"_root.ar.root.publications.base"
                        LEFT OUTER JOIN LATERAL (
                   SELECT row_to_json(
                                  (SELECT "_e"
                                   FROM (SELECT
"_root.ar.root.publications.or.author.ar.author.publications"."publications" AS
"publications",

"_root.ar.root.publications.or.author.base"."full_name"                      AS
"full_name") AS "_e")
                          ) AS "author"
                   FROM (SELECT *
                         FROM "public"."author"
                         WHERE (

("_root.ar.root.publications.base"."author_id") = ("author_id")
                                   )
```

```
                             LIMIT 1) AS
"_root.ar.root.publications.or.author.base"
                                LEFT OUTER JOIN LATERAL (
                           SELECT coalesce(json_agg("publications"), '[]') AS
"publications"
                           FROM (SELECT row_to_json(
                                                (SELECT "_e"
                                                 FROM (SELECT
"md5_c3c0706a8898cc1a5785e9177f2658da__root.ar.root.publications.or.author.ar.a
uthor.publications.or.article"."article" AS "article") AS "_e")
                                     ) AS "publications"
                                 FROM (SELECT *
                                        FROM "public"."publications"
                                       WHERE (
                                                       (

"_root.ar.root.publications.or.author.base"."author_id"
                                                             ) = ("author_id")
                                                 )) AS
"md5_fa9de4bfd2936a5a64cb7fc66180aa42__root.ar.root.publications.or.author.ar.a
uthor.publications.base"
                                        LEFT OUTER JOIN LATERAL (
                                   SELECT row_to_json(
                                                    (SELECT "_e"
                                                     FROM (SELECT
"md5_fc63b0bec1fa4ceec72a5b68f2d6faa6__root.ar.root.publications.or.author.ar.a
uthor.publications.or.article.base"."title"    AS "title",

"md5_fc63b0bec1fa4ceec72a5b68f2d6faa6__root.ar.root.publications.or.author.ar.a
uthor.publications.or.article.base"."abstract" AS "abstract") AS "_e")
                                         ) AS "article"
                                   FROM (SELECT *
                                          FROM "public"."article"
                                         WHERE (
                                                          (

"md5_fa9de4bfd2936a5a64cb7fc66180aa42__root.ar.root.publications.or.author.ar.a
uthor.publications.base"."article_id"
                                                                ) = ("article_id")
                                                      )
                                         LIMIT 1) AS
"md5_fc63b0bec1fa4ceec72a5b68f2d6faa6__root.ar.root.publications.or.author.ar.a
uthor.publications.or.article.base"
                                   ) AS
"md5_c3c0706a8898cc1a5785e9177f2658da__root.ar.root.publications.or.author.ar.a
uthor.publications.or.article"
                                                            ON ('true')) AS
"_root.ar.root.publications.or.author.ar.author.publications"
```

```
                       ) AS
"_root.ar.root.publications.or.author.ar.author.publications" ON ('true')
                   ) AS "_root.ar.root.publications.or.author" ON ('true')) AS
"_root.ar.root.publications"
         ) AS "_root.ar.root.publications" ON ('true')) AS "_root";
```

## Cypher queries

```
// find authors who are often co-authors
MATCH (a1:Author)-[:AUTHORED]->(:Title)<-[:AUTHORED]-(a2:Author)
WITH a1, a2, count(a1) AS coauthorCount
WHERE coauthorCount > 5
RETURN a1.Name, collect(a2.Name) AS FrequentCoAuthors

// find authors who are often associated with the same category
MATCH
(a1:Author)-[:AUTHORED]->(:Title)-[:IS_CATEGORIZED_AS]->(c:Category_List)<-[:IS
_CATEGORIZED_AS]-(:Title)<-[:AUTHORED]-(a2:Author)
RETURN a1.Name, c.Category_List, collect(a2.Name) AS Co_Authors

CALL gds.pageRank.stats('Author', {
 maxIterations: 20,
 dampingFactor: 0.85
})
YIELD centralityDistribution
RETURN centralityDistribution.max AS max
```