Using graph databases in source code plagiarism detection

Matija Novak, Iva Levak
Faculty of Organization and Informatics
University of Zagreb
Pavlinska 2, 42000 Varaždin
matija.novak, iva.levak@foi.unizg.hr

Abstract. *Modern plagiarism detection tools calculate the percentage of the similarity between two given source code files. In academia, the process of checking for potential plagiarized students solutions can be challenging in terms of resources due to the large number of combinations between many students. In such conditions, the reliability of plagiarism detection tools may be put to risk. Every plagiarism detection tool produces a similarity report as files containing the results of the analysis for each pair of analyzed source code files. While such a report is useful for a one-time checking, sometimes it is needed to store the result data for future use. In our previous work, the results were stored in a relational database and a list of relevant queries was defined for meaningful analysis. Nevertheless, the large number of pair-wise impacts the storage and query execution speeds. In this paper, we present a solution to this problem by importing the similarity analysis data into a graph database and evaluate the difference in the query execution speed between a graph and a relational database.*
Keywords. plagiarism, graph databases, similarity detection

1 Introduction

In academia, students are required to do homework assignments as part of their course evaluation. One problem that teachers have with homework assignments is plagiarism. Plagiarism is "the act of taking the writings of another person and passing them off as one's own" according to (Encyclopædia Britannica Inc, 2015). There is a similar problem called collusion which means "working together to produce assessed work in circumstances where this is forbidden" (Barrett and Cox, 2005). The main difference is that in collusion the original author knows and participates in the process of cheating. While there are many definitions what constitutes plagiarism in this paper the definition from (Novak, 2020) will be used which is defining specifically source-code plagiarism and it seems the most accurate in this case: "Plagiarism, in programming assignments, is the act of taking a significant amount of source-code parts (up to the entire source-code) from other students or from the Internet and using it without noting its origin. A 'significant amount' means that the similarity between two solutions of a programming assignment is high enough that an expert (teacher, ethical board, etc.) considers specific student work as sufficiently 'real' plagiarism to accuse the student of plagiarism.".

The main problem when dealing with plagiarism is how to find out that plagiarism is taking place. In small classrooms this is not a big issue just by grading the assignments the teacher notices similarity and can act accordingly. But in big classrooms (100 or more students) it is impossible to remember every solution and compare them. In such situations, similarity detection tools are a significant assistance. Such tools are often called plagiarism detection tools yet it's not really what they perform rather for what purpose they are used. So-called plagiarism detection tools calculate the percentage of the similarity between two given files. Some tools are for text and some are specialized for source-code. A good comparison can be found in (Ragkhitwetsagul et al., 2016; Novak et al., 2019). As already mentioned, in this paper the focus is on source-code similarity since the chosen courses are programming courses.

Plagiarism detection tool produces a similarity reports as files and while such a report is useful for a one-time checking, sometimes it is needed to store the result data for future use. In our previous work, the results were stored in a relational database and list of relevant queries was defined for meaningful analysis. Nevertheless, the large number of pair-wise impacts the storage and query execution speeds. In this paper, we present a solution to this problem by importing the similarity analysis data into a graph database and evaluate the difference in the query execution speed between a graph and a relational database.

Section 2 gives the related work. In section 3 the plagiarism detection process is presented and more details about concrete problem are given. Section 4 describes the methodology and section 5 describes the transition to graph database. Section 6 presents and discusses the query execution results in both databases and section 7 concludes.

2 Related work

In works like (Joy and Luck, 1999; Martins et al., 2015; Đurić and Gašević, 2013) various similarity detections are performed with the purpose of detecting plagiarism in academia source-code assignments. The first works date back to 80's like (Donaldson et al., 1981). But, when looking for plagiarism detection engines or similarity detection engines that use graph database, there is not much work done. There is a paper by (Arora et al., 2021) where they use graph databases to visualize the code structure so it can be analysed. Likewise, in (Tyagi et al., 2022) they are "converting a Java program into a specialized dependency graph". On the other hand there are a lot of papers like (Esser and Fahland, 2021; Timón-Reina et al., 2021; Šestak et al., 2021) and books (e.g. (Robinson et al., 2015)) about graph databases and their usage in various areas (e.g. Facebook's Open Graph, Google's Knowledge Graph, Twitter's FlockD) as stated by (Miller, 2013). In this paper we want to apply graph databases in the context of plagiarism detection and see if we can get good results. In difference to the previously mentioned work in this paper we use the typical similarity detection tools (e.g. Sherlock, jPlag) for detection and graph database in the analysis's phase. More details about the tools are given in the next section.

3 Plagiarism detection process

In (Đurić and Gašević, 2013; Kermek and Novak, 2016) the process of plagiarism detection is described, which can be divided into four main phases: preparation phase, preprocessing phase, detection phase and result analysis phase. The preparation phase is usually not complicated, especially if students submit assignments in Learning Management Systems like Moodle, it includes downloading the code and passing it to the preprocessing phase. Preprocessing phase deals with cleaning the data before detection and an in-depth analysis of the effectiveness of preprocessing was done in (Kleiman and Kowaltowski, 2009; Đurić and Gašević, 2013; Novak, 2020). Afterwards, the similarity is calculated using tools like SIM (Grune and Huntjens, 1989), Sherlock (Joy and Luck, 1999), JPlag (Prechelt et al., 2002) etc. These tools produce output in various forms, but usually the data is displayed in a table form (Prechelt et al., 2002; Grune and Huntjens, 1989) or graph form (Joy and Luck, 1999; Novak et al., 2021). This process normally takes a while depending on the number of submissions and the code lengths of each submission. At last, the data is analysed by the teacher by looking at the highest similarities mainly in side-by-side comparison and deciding if plagiarism is taking place.

While our process is the same in the first three phases the analysis phase differs. In our case, students can submit their assignments at different points of time usually before an exam. According to the faculty regulations, a student can retake the exam (if he does not pass) up to eight times in four years. Therefore students from different academic years submit their projects before each exam. In order to make sure that students did not use the projects or parts of the projects from students that already passed, the course plagiarism detection needs to be performed before each exam. The comparisons are done between student projects from the same academic year since they have the same assignment. The project assignment is to build up a full-featured web application that in the end has around 1500-5000 lines of code (LOC). In order to not repeat detections that were already done, the data is stored in a relational database. Then before each exam, a query is performed to get similarities larger than 30% for students that are taking the exam. If high similarities occur, then projects similarities are analysed manually and if plagiarism is taking place the student can not take the exam and is reported to the ethical board.

Conducting plagiarism analysis happens right before every exam for each applied student. Number of examined students can vary from one student to roughly one hundred. The issue here is that over the years the database became quite large (over 1.200.000 data) and the query executions became slow. By looking at the MySQL process status it was noticed that sometimes a query would take up to 1 minute to execute. This can be problematic if 50 students apply for an exam so it takes up too much time to execute all queries for all applied students. Naturally, the execution not only depends on the number of students that applied but also on the server load, the number of plagiarised matches, etc. It is not possible to influence the server load, also the number of plagiarised matches is not something that can be changed and many other factors are out of our reach. But maybe the query executions can be improved by cleaning up the database of old data, which didn't solve the problem. Afterwards, we tried to optimize the queries so that only one complex query is submitted instead of multiple queries. While it seemed better at first, by monitoring the query times in MySQL process status we noticed that some queries still took around 40 seconds. The query that was used looked like this:

```
SELECT nameP,similarity,fileType
FROM (SELECT
  CONCAT(‘username1‘,’-’,‘username2‘)
    as nameP,
    ‘username1‘ , ‘username2‘ ,
    ‘fileType‘ ,
    SUM( ‘similarity‘ ) as ‘similarity‘
  FROM ‘matches‘
  WHERE
  (username1=’studentA’
    OR username2=’studentA’)
  GROUP BY ‘username1‘ , ‘username2‘,
  ‘fileType‘
```

```
order by similarity DESC) as temp
WHERE ‘similarity‘ > 20
```

Mentioned query returns information about which student has significantly high similarity to observed student. Nested query produces results which includes observed student and his observed pairs and their total of similarities by file type.

It might be that the query can be further optimised but since there is a work about graph databases that shows their advantage over the relational database like (Cheng et al., 2019) the idea is to migrate the data to a graph database and see if the query execution will be significantly faster.

4 Methodology

To verify if graph database is a good choice in this situation we performed an experiment. To have comparable results the experiment was done for both databases on the same machine with 4GB of RAM and 4CPU cores. We are aware that it's recommended to have at least 8GB of RAM for Neo4j but we wanted to perform experiment in environment which can be used by an individual teacher. We are aware that needs for used tools are not met, but if we manage to improve the times on such hardware then it is the expectation that the times will be even better on a server.

In the experiment we will measure time to execute the query for 1 student, 33 students and two times the same 33 students which makes a total of 66 students. Also, further improvement of the MySQL query was performed and at last the query used for testing in MySQL looks like this:

```
SELECT CONCAT(‘username1‘,’-’,‘username2‘)
    AS imeP,
    SUM(similarity) as similarity,fileType
FROM matches
WHERE (username1=’studentA’
   OR username2=’studentA’)
GROUP BY ‘username1‘ ,
   ‘username2‘, ‘fileType‘
HAVING SUM(similarity) > 20
ORDER BY similarity DESC
```

First of all, query is filtering all data by examined student username. Query then produces totals of similarities between examined student and its pair student. Finally, query filters only the results in which similarity is greater than some number, in this case twenty.

Before conducting the experiment, all data needs to be transitioned to the graph database.

5 Transition to graph database

In order to transition to a graph database the data was exported from a relational database and stored

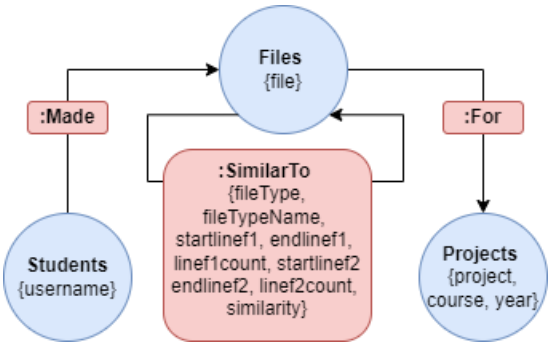

Figure 1: Graph Database schema

first as a CSV file. CSV file was then used for importing data into graph database. Neo4j [1] was used for graph database and installing it was very simple but the main problem was importing all the data into the database. The data for import was stored at $/var/lib/neo4j/import$ directory so it can be loaded with:

```
LOAD CSV WITH HEADERS FROM
"file:///matches.csv" as line
CREATE(:Plagijat{file1: line.file1,
file2:line.file2,
similarity:toInteger(line.similarity),
username1:line.username1,
username2:line.username2,
project:line.projekt,
course: line.predmet,
year:toInteger( line.godina)});
```

This was the first attempt which did not actually create graph rather it created independent nodes where one node was corresponding to one row in the relational database. Keywords WITH HEADERS allowed us to load CSV files with header row. To create graph in database we used next database schema as presented in Figure 1.

In database schema there are three types of nodes: Students, Files and Project which are connected with relationships :Made and :For. Main relationship :SimilarTo is recursive and defines file pairs similarities. In relation with created database schema, loading of data was with next query.

```
LOAD CSV WITH HEADERS
FROM "file:///load1.csv" AS line
WITH line
  MERGE (f1:Files {file:line.file1})
  MERGE (f2:Files {file:line.file2})
  MERGE (p:Projects{project:line.projekt,
    year:toInteger(line.godina),
    course:line.predmet})
WITH f1, f2, p, line
  MERGE (f1)-[:For]->(p)<-[:For]-(f2)
WITH f1, f2, line
```

[1] https://neo4j.com/

```
MERGE (f1)-[:SimilarTo
  {fileType:line.fileType,
  fileTypeName:line.fileTypeName,
  startlinef1:line.startlinef1,
  endlinef1:line.endlinef1,
  linef1count:line.linef1count,
  startlinef2:line.startlinef2,
  endlinef2:line.endlinef2,
  linef2count:line.endlinef2,
  similarity:line.similarity}]->(f2)
```

Rather then CREATE, clause MERGE was used as it intends to create nodes and relationships that don't already exist in database. Data in CSV file were divided in sequences of 100 000 lines per file because of heap memory. With given query, two of three mentioned nodes were created, Files and Projects, and two of three relationships, :For and :SimilarTo. Nodes and relationships are created with properties corresponding to schema in Figure 1. Relationship :For defines which file belongs to which project, and relationship :SimilarTo defines plagiarism detection results between two files. In consequence of big data, student usernames were loaded separately. First, usernames with corresponding file names were excluded from CSV files and loaded with next query.

```
LOAD CSV WITH HEADERS
FROM "file:///studentFiles.csv" AS line
WITH line
  MERGE (f:Files{file:line.file})
  MERGE (s:Students
    {username:line.username})
WITH f,s
  MERGE (s)-[:Made]->(f)
```

Query above was used for importing student usernames as node named Students and connecting usernames and matched files with relationships :Made. To do the testing we used Node since it has packages for both MySQL and Neo4j in the repositories:

```
sudo npm -g install mysql2
sudo npm -g install neo4j-driver
```

6 Results and discussion

For tests, we have performed the same queries on a relational database and on a graph database. Not exactly the same, since they used different query types but with the same intent, conditions and data comparisons. The idea was to get all similarities bigger than some percentage for a particular student. While executing queries, the time was measured for how long it takes to get a result response.

In Table 1 the average execution time is presented. The test was performed with only 1 student, then with 33 different students and then with 2 times 33 students which means that every student was queried twice. The total amount of data was 1.200.000 in both databases. The first attempt with was importing data into GraphDB without much considering the graph and as one can see the GraphDB query is slower than the MySQL optimized query as there is no graph to work with. The query in the graph database looked like this:

```
MATCH(p:Plagijat) WHERE
  (p.username1="studentA"
    OR p.username2="studentA")
  WITH {type:p.type,
       user1:p.username1,
       user2:p.username2,
       suma:SUM(p.similarity)}
  AS u WHERE u.suma>20
  RETURN u
```

As database didn't have relationships, this query is very similar to MySQL query, with disctincion of not having to explicitly specify grouping of data. Grouping is in fact formed from the non-aggregation columns in scope when aggregation function like SUM is used. Query searches for all data with specified student username and returns data with sum of similarities of students per file type.

As mentioned, second experiment was to create a real graph database with relationships. In this case the benefit is in having a graph representation to analyse one student relations to others. This time the query looked like this:

```
MATCH(s:Students{username:"studentA"})
  --()-[st:SimilarTo]->()--(s1:Students)
WITH {type:st.type,
  user1:s.username,
  user2:s1.username,
  suma:SUM(toInteger(st.similarity))}
AS u WHERE u.suma>20 RETURN u
```

Previous query searches for all relationships :SimilarTo from student with username "studentA" to all students. Although not explicitly written, in query there are used relationship :Made and nodes Files. Full MATCH clause would look like:

```
MATCH(s:Students{username:"studentA"})
  -[:Made]->(:Files)-[st:SimilarTo]
  ->(:Files)-[:Made]->(s1:Students)
```

Query firstly selects node :Students with username "studentA" then finds all the files that student made, and secondly finds files of another students which files where compared to files of "studentA". Lastly it returns calculated similarity between two students.

One can see from Table 1 that with added relationships, execution time is improved. For better comparison of MySQL optimized query and Neo4j query with relationships, query execution times for one student are presented as graph in Figure 2. In Figure 3 the results are presented for executing a queries for 33 students,

Table 1: Results summary

Tool	Students	AVG Total Time
MySQL	1	0.496s
Neo4j with one node	1	1.628s
Neo4j with relationships	1	1.092 s
MySQL	33	18.525s
Neo4j with one node	33	19.172s
Neo4j with relationships	33	2.468s
MySQL	2*33	38.101s
Neo4j with one node	2*33	41.718s
Neo4j with relationships	2*33	2.679s

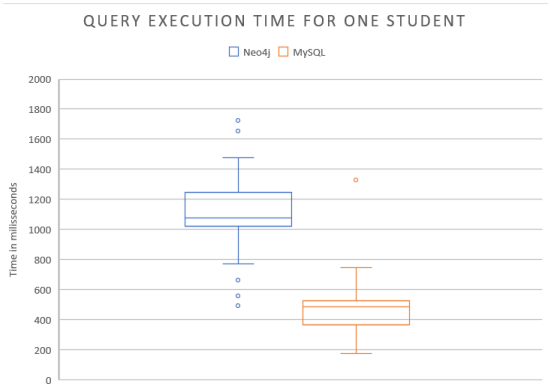

Figure 2: Query execution time for 1 student

and in figure 4 the results are presented for executing a queries for 66 students. From this analysis we conclude that MySQL performed better when executing only one query but when the number of queries is larger then Neo4j starts to outperform MySQL.

One of the big advantages of using graph database is that tools like Gephi are not needed, as used in (Novak et al., 2021) where the authors display the benefits of graph representation of student file similarities. With graph database one can directly have a graph representation without any need for other tools. In Figure 5 we can see connections of selected student with other students.

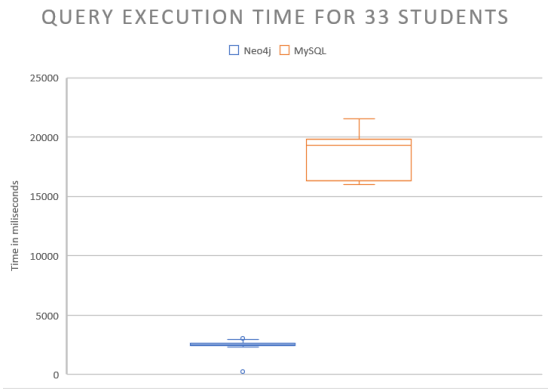

Figure 3: Query execution time for 33 students

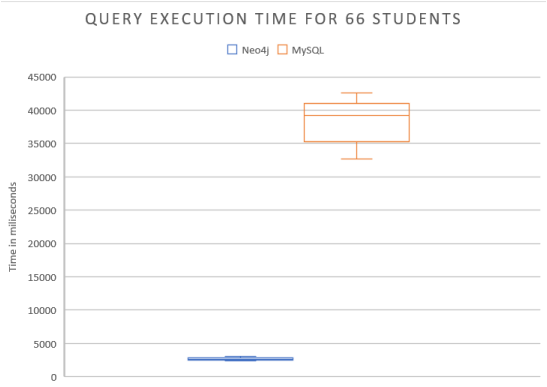

Figure 4: Query execution time for 66 students

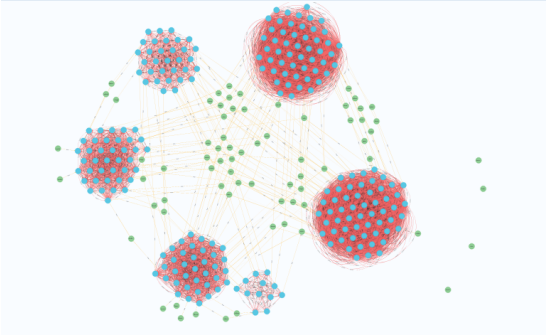

Figure 5: Node connections for one username in Neo4j

7 Conclusion

In this paper, we have shown that graph databases provide a lot of benefits for analysing data in the context of plagiarism detection. Firstly, it represents data connections which can be used for analysis so there is no need for installation and configuration of additional tools for creating graphs. Furthermore, our testing results show that with the correct graph database scheme, executing presented queries and getting the result set of data from a graph database outperforms the relational database when the number of queries increases. With only one query MySQL had an average of 0.495 seconds execution time while Neo4j had 1.098 seconds execution time. On the other hand with 33 students the average execution time of MySQL queries increased to 18.525 seconds while average execution time of Neo4j queries was only 2.468 seconds.

In future works, the idea is to expand research by additional testing on created graph database to determine the real impact of graph databases in the process of plagiarism detection. Moreover, additional tests have to be performed on better hardware with different data in order to have a better understanding and to be able to generalize the findings. Current setup was used intentionally since this is a hardware configuration that most teacher should have available. Based on current results, although experiment was performed on

lower-end hardware and because Neo4j outperformed MySQL, we expect that with better hardware the differences will even more in favor of Neo4j since it demands at least 8GB of RAM.

References

Arora, R., Motilal Maurya, A., and Sharma, Y. (2021). Application of java relationship graphs (jrg) to plagiarism detection in java projects: A neo4j graph database approach. In *2021 The 4th International Conference on Software Engineering and Information Management*, pages 46–51.

Barrett, R. and Cox, A. L. (2005). At least they're learning something: the hazy line between collaboration and collusion. *Assessment & Evaluation in Higher Education*, 30(2):107–122.

Cheng, Y., Ding, P., Wang, T., Lu, W., and Du, X. (2019). Which category is better: benchmarking relational and graph database management systems. *Data Science and Engineering*, 4(4):309–322.

Đurić, Z. and Gašević, D. (2013). A Source Code Similarity System for Plagiarism Detection. *The Computer Journal*, 56(1):70–86.

Donaldson, J. L., Lancaster, A.-M., and Sposato, P. H. (1981). A plagiarism detection system. *ACM SIGCSE Bulletin*, 13(1):21–25.

Encyclopædia Britannica Inc (2015). Plagiarism. Accessed on 2015-09-10 Available at http://www.britannica.com/topic/plagiarism.

Esser, S. and Fahland, D. (2021). Multi-dimensional event data in graph databases. *Journal on Data Semantics*, 10(1):109–141.

Grune, D. and Huntjens, M. (1989). Het detecteren van kopieen bij informatica-practica. *Informatie*, 31(11):864–867.

Joy, M. and Luck, M. (1999). Plagiarism in programming assignments. *IEEE Transactions on Education*, 42(2):129–133.

Kermek, D. and Novak, M. (2016). Process Model Improvement for Source Code Plagiarism Detection in Student Programming Assignments. *Informatics in Education*, 15(1):103–126.

Kleiman, A. and Kowaltowski, T. (2009). Qualitative Analysis and Comparison of Plagiarism-Detection Systems in Student Programs - Technical Report IC-09-08. Technical report, Institu to de Computação, Universidade Estadual de Campinas.

Martins, V. T., Henriques, P. R., and da Cruz, D. (2015). An AST-based Tool, Spector, for Plagiarism Detection: The Approach, Functionality, and Implementation. *Communications in Computer and Information Science*, 563:153–159.

Miller, J. J. (2013). Graph database applications and concepts with neo4j. In *Proceedings of the southern association for information systems conference, Atlanta, GA, USA*, number 36 in 2324.

Novak, M. (2020). *Effect of source-code preprocessing techniques on plagiarism detection accuracy in student programming assignments*. PhD thesis, Sveučilište u Zagrebu. Fakultet organizacije i informatike.

Novak, M., Joy, M., and Kermek, D. (2019). Source-code similarity detection and detection tools used in academia: A systematic review. *ACM Transactions on Computing Education*, 19(3):27:1–27:37.

Novak, M., Joy, M. S., and Mirza, O. M. (2021). Improved plagiarism detection with collaboration network visualization based on source-code similarity. In *2021 IEEE Technology & Engineering Management Conference-Europe (TEMSCON-EUR)*, pages 1–6. IEEE.

Prechelt, L., Malpohl, G., and Philippsen, M. (2002). Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8(11):1016–1038.

Ragkhitwetsagul, C., Krinke, J., and Clark, D. (2016). Similarity of Source Code in the Presence of Pervasive Modifications. In *IEEE 16th International Working Conference on Source Code Analysis and Manipulation*, pages 117–126. IEEE.

Robinson, I., Webber, J., and Eifrem, E. (2015). *Graph databases: new opportunities for connected data*. " O'Reilly Media, Inc.".

Timón-Reina, S., Rincón, M., and Martínez-Tomás, R. (2021). An overview of graph databases and their applications in the biomedical domain. *Database*, 2021.

Tyagi, D., Arora, R., and Sharma, Y. (2022). Application of java relationship graphs to academics for detection of plagiarism in java projects. In *ICT Systems and Sustainability*, pages 761–772. Springer.

Šestak, M., Heričko, M., DruÅ¾ovec, T. W., and Turkanovič, M. (2021). Applying k-vertex cardinality constraints on a neo4j graph database. *Future Generation Computer Systems*, 115:459–474.