Comparison of relational and graph databases

Authors:
Radosław Bielecki, 64977
Paweł Bryzek, 64980

1. Abstract

We will be comparing Relational Databases and Graph databases on data from kaggle competition: Data Science for Good: CareerVillage.org - Match career advice questions with professionals in the field.
We will compare specifications of both databases and execution speed of different operations and for different queries.

2. Data

https://www.kaggle.com/competitions/data-science-for-good-careervillage/data

CareerVillage.org has provided several years of anonymized data and each file comes from a table in their database. The data comes in several .csv files:

- answers.csv: Answers get posted in response to questions. Answers can only be posted by users who are registered as Professionals. However, if someone has changed their registration type after joining, they may show up as the author of an Answer even if they are no longer a Professional. This dataset has 51123 rows and 5 columns:
  - answers_id - contains a value used for identifying the answer, example value: 4e5f01128cae4f6d8fd697cec5dca60c
  - answers_author_id - contains a value used for identifying the author that gave the answer, example value: 36ff3b3666df400f956f8335cf53e09e
  - answers_question_id - contains a value used for identifying the question example value: 332a511f1569444485cf7a7a556a5e54
  - answers_date_added - contains a value specifying at what time and what day was the answer posted, example value: 2016-04-29 19:40:14 UTC+0000
  - answers_body - contains a plain text with the content of the answer
- comments.csv: Comments can be made on Answers or Questions. We refer to whichever the comment is posted to as the "parent" of that comment. Comments can be posted by any type of user. This dataset has 14966 rows and 5 columns:
  - comments_id - contains a value used for identifying the comment, example value: f30250d3c2ca489db1afa9b95d481e08
  - comments_author_id - contains a value used for identifying the author that wrote the comment, example value: 9fc88a7c3323466dbb35798264c7d497

    - comments_parent_content_id - contains a value used for identifying the parent of the comment, example value: b476f9c6d9cd4c50a7bacdd90edd015a
    - comments_date_added - contains a value specifying at what time and what day was the comment posted, example value: 2019-01-31 23:39:40 UTC+0000
    - comments_body -contains a plain text with the content of the comment

  As we can clearly see the structure of comments.csv and answers.csv is very similar

- emails.csv: Each email corresponds to one specific email to one specific recipient. The frequency_level refers to the type of email template which includes immediate emails sent right after a question is asked, daily digests, and weekly digests. This file has 1850102 rows and 4 columns:
    - emails_id - contains a value used for identifying the email, example value: 2337714
    - emails_recipient_id - contains a value used for identifying the email recipient, example value: 0c673e046d824ec0ad0ebe012a0673e4
    - emails_date_sent - contains a value specifying at what time and what day was the email sent, example value: 2018-12-07 01:05:40 UTC+0000
    - emails_frequency_level - contains information about the frequency level such as daily or weekly mails. Example value: email_notification_daily

- group_memberships.csv: Any type of user can join any group. There are only a handful of groups so far. This dataset has 1039 rows and 2 columns:
    - group_memberships_group_id - contains an id specifying the id of a group the user belongs to, example value: eabbdf4029734c848a9da20779637d03
    - group_memberships_user_id - contains an id specifying the id of the user that belongs to a given group , example value: 9a5aead62c344207b2624dba90985dc5

- groups.csv: Each group has a type, which specifies the purpose of the group. For privacy reasons the group names are off in the dataset. This datasets contains 49 rows and 2 columns:
    - groups_id - contains an id specifying the id of a group, example value: eabbdf4029734c848a9da20779637d03

    - groups_group_type - contains the type of the group, example value: youth program

- matches.csv: Each row tells you which questions were included in emails. If an email contains only one question, that email's ID will show up here only once. If an email contains 10 questions, that email's ID would show up here 10 times. This dataset has over 4 million rows and 2 columns:
    - matches_email_id - contains an id specifying the id of a match, example value: 1635498
    - matches_question_id - contains and id of question which was included in given email, example value: 332a511f1569444485cf7a7a556a5e54

- professionals.csv: Users with experience in some industry are called "Professionals". With 28 thousand professionals and 5 columns:
    - “professionals_id” - contains a value used for identifying user,
    - “professionals_location” - can contain location of user,
    - “professionals_industry” - can contain industry that professional has knowledge of,
    - “professionals_headline” - can contain title of professional ex. “Ph. D”,
    - “professionals_date_joined” - date when professional joined site.

- questions.csv: Questions get posted by students. Sometimes they're very advanced. Sometimes they're just getting started. There are almost 24 thousand questions, with 5 columns:
    - “questions_id” - contains a value used for identifying question,
    - “questions_author_id” - contains a value used for identifying author of question,
    - “questions_date_added” - date of question added,
    - “questions_title” - string with title of question,
    - “questions_body” - string with question,

- school_memberships.csv: Just like group_memberships, but for schools instead. With 5,6 thousand rows and 2 columns :
    - “school_memberships_school_id” - contains a value used for identifying school,
    - “school_memberships_user_id” - contains a value used for identifying user

- students.csv: Students are the most important people on CareerVillage.org. They tend to range in age from about 14 to 24. With almost 31 thousand students and 3 columns:
    - “students_id” - contains a value used for identifying student,
    - “students_location” - Sometimes contains location of student,
    - “students_date_joined” - Date when student joined site.

- tag_questions.csv: Every question can be hashtagged. The hashtag-to-question pairings are tagged, and put into this file. With 76,5 thousand rows and 2 columns:
    - “tag_questions_tag_id” - number (tag) that the question was tagged with,
    - “tag_questions_question_id” - contains a value used for identifying question.

- tag_users.csv: Users of any type can follow a hashtag. This shows which hashtags each user follows. With 136,6 thousand rows and 2 columns:
    - “tag_users_tag_id” - value (tag) that user follows,
    - “tag_users_user_id” - contains a value used for identifying user.

- tags.csv: Each tag gets a name. With 16 thousand different tags and 2 columns:
    - “tags_tag_id” - contains a number used for identifying tag,
    - “tags_tag_name” - string with name of the tag.

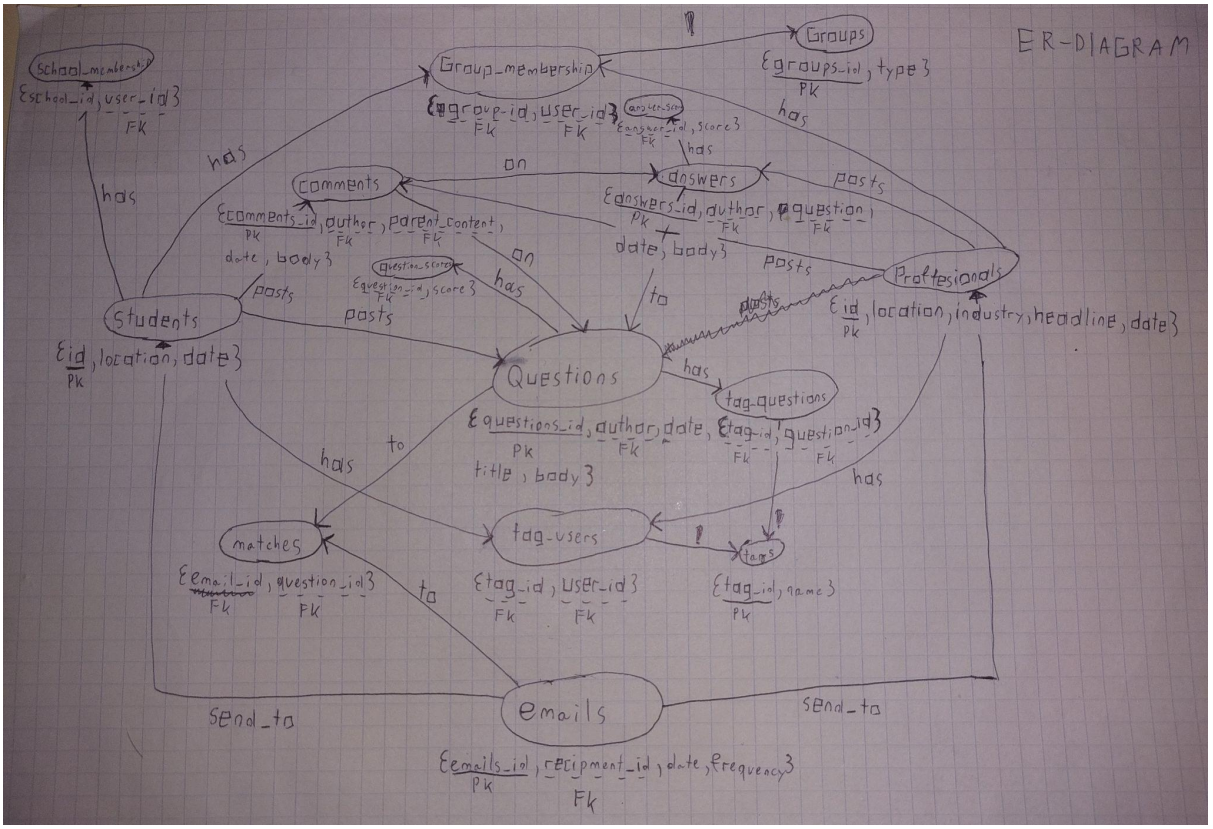
- question_scores.csv: Scores for each question. With almost 24 thousand different questions and 2 columns:
    - “id” - contains a value used for identifying question,
    - “score” - a number that question was scored with, from 0 to 125.

- answer_scores.csv: Scores for each answer. With over 50 thousand rows and 2 columns
    - “id” - contains a value used for identifying answer,
    - “score” - a number that the answer was scored with, from 0 to 30.

3. Used Technologies

Neo4j - as a graph database management system
MSSQL - as a relational database management system

4. ER-Diagram of database:



5. Creating Neo4j desktop database:

To create an optimal database in Neo4j we need to assign Unique constraints to data rows with primary keys and we need to assign all data values with their proper data types.

In Neo4j for constraints we use, ex:

```
CREATE CONSTRAINT UniqueStudent ON (s:Students) ASSERT s.students_id IS UNIQUE;
```

For data types we use, ex:
datetime(), toInteger().

While creating database in Neo4j it is good habit to first check if the data loads correctly, ex:

```
LOAD CSV WITH HEADERS FROM 'file:///main/students.csv' AS row

WITH row.students_id AS students_id,
```

```
row.students_location AS students_location,

datetime(replace(replace(row.students_date_joined, " UTC", ""), " ", "T")) AS
students_date_joined

RETURN students_id, students_location, students_date_joined

LIMIT 5;
```

And only after that load csv to the database, ex.

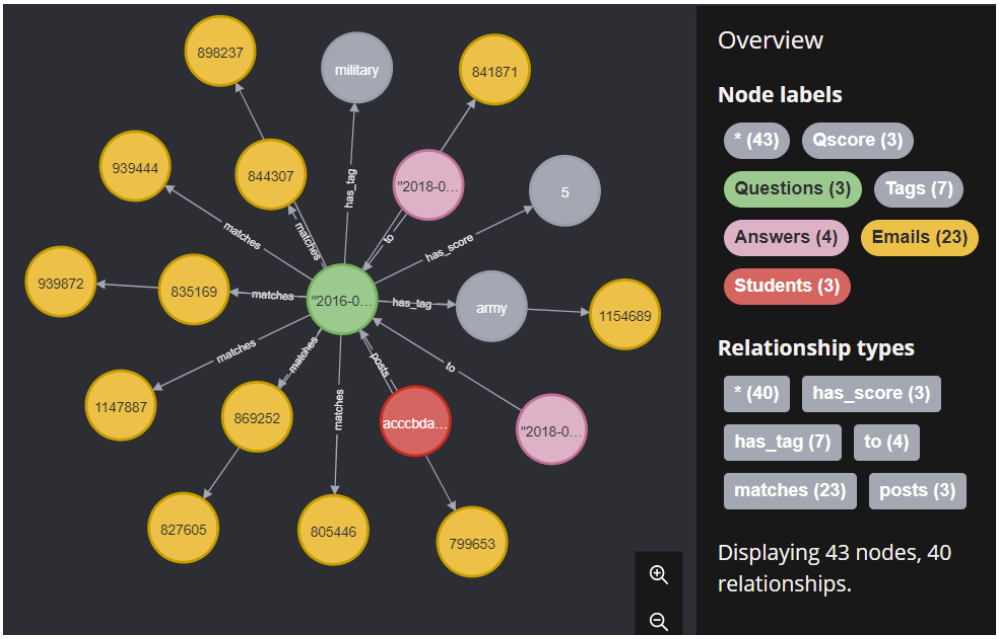
```
auto USING PERIODIC COMMIT

LOAD CSV WITH HEADERS FROM 'file:///main/students.csv' AS row

WITH row.students_id AS students_id,

row.students_location AS students_location,

datetime(replace(replace(row.students_date_joined, " UTC", ""), " ", "T")) AS
students_date_joined

MERGE (s:Students {students_id: students_id})

  SET s.students_location = students_location, s.students_date_joined = students_date_joined

RETURN count(s);
```

After creating all nodes that we need for given relationship, we can create relationships, like:

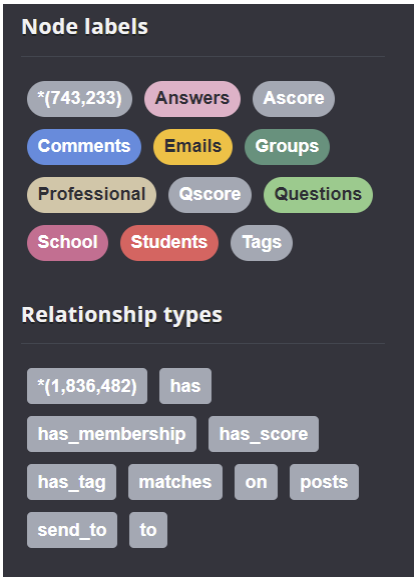
```
MATCH (s:Students {students_id:s.students_id})

MATCH (q:Questions {questions_author_id:s.students_id})

MERGE (s)-[:posts] ->(q)
```

Interesting data that we get from Noe4j right away after creating nodes or relationships is how many Nodes or Relationships were we able to create. If Neo4j is creating Relationship for data that is only in one of two nodes, rather than giving error like in SQL, he will skip this data.

Example of graph for Questions and all its relationships:



Whole database look like that:



Problems with Neo4j:

- To create a database a lot of code needs to be written, for every relation and node we need to write different queries,
- There is a way to save whole database in desktop Neo4j, but this way is hidden in documentation and not always works,
- We were using the Community version which has some limitations, like the amount of data possible to be stored in a Node. For this reason we had to LIMIT loaded emails data to 500000 entities,
- Time for all loading queries grows with correlation to growing amount of rows in data,

- Neo4j is very hardware excessive, which also brought problems with data loading times.

Advantages:

- Creating nodes and relationships with Neo4j takes longer time than in SQL, but after creating them, it finds all high dimension relationships much faster,
- Graphs from Neo4j are very easy to read,
- For some complicated queries need much less code than SQL.

6. Creating sql database

First, before creating any tables in SQL databases it is a good habit to check the data for unwanted characters - for example in our data in Answers table there is a column ‘answers body’, which contains content of the answer. Users can put anything there so it is good to make sure there are no additional semicolons or carriages there, which may make it harder for sql to interpret the data. Additionally by analyzing data, one should check the types of input data, because in MSSQL it's important to set a proper type for every column. For example one of the mistakes we made at the beginning was setting the location data type as nvarchar(50), which later caused some mistakes because some location names exceeded 50 characters.
After importing the data it’s important to create relations between tables. In case of this data, it was necessary to create two additional tables:

1) Table users which contain the id of all users - both professionals and students. It was important, because one foreign key can correspond only to one primary key, so there was no way of creating relations between a few tables - for example both students and professionals can post comments, and the author of the comment is a foreign key in a table.
2) Table posts which contain id of all posts - both questions and answers. This was necessary for the same reason - comments can be posted to both answers and questions, and information about what is being commented is stored and is a foreign key.

These tables were created by joining columns from other tables into one. Here is a code used for that:

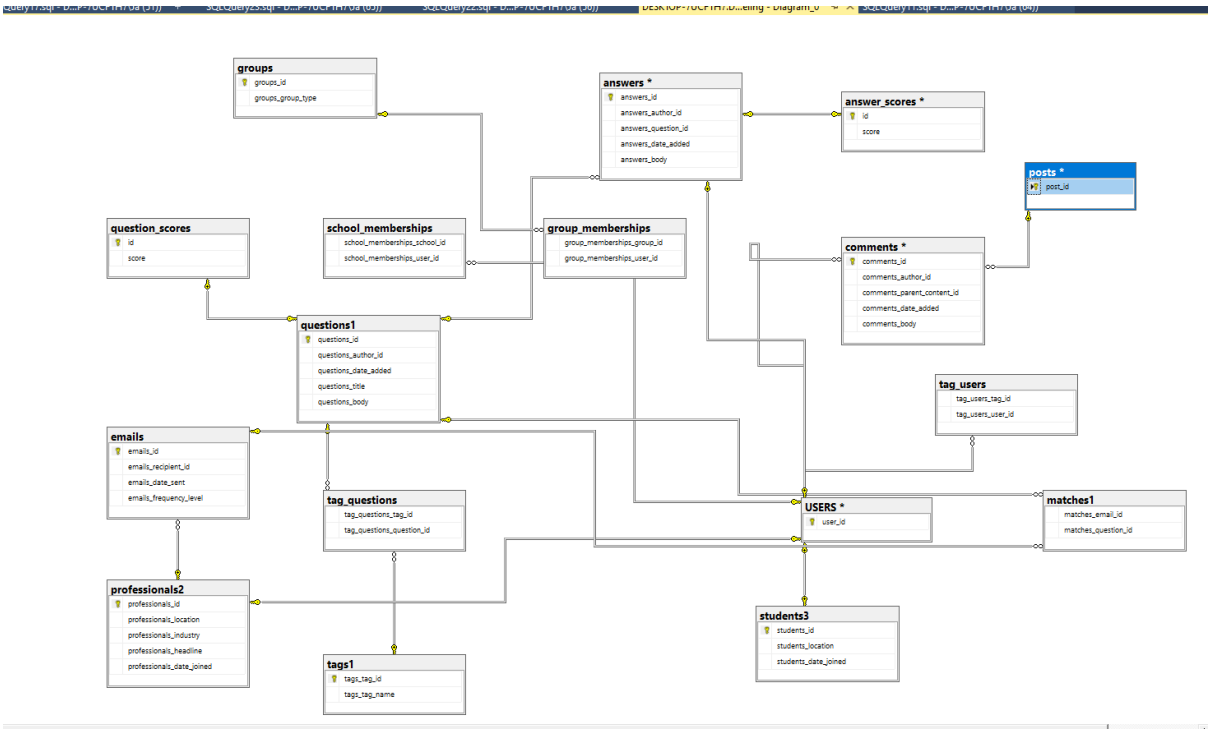
```
SELECT professionals2.professionals_id as user_id INTO USERS
FROM professionals2

UNION

SELECT students3.students_id
FROM students3

ALTER TABLE USERS ADD PRIMARY KEY(user_id)
```

```
SELECT questions1.questions_id as post_id INTO posts
FROM questions1

UNION

SELECT answers.answers_id
FROM answers

ALTER TABLE pots ADD PRIMARY KEY(posts)
```

During the creation of the relations, it turned out that part of the data was corrupted, some answers and comments did not have an author. The amount of data like that was very small, so we decided to delete corrupted rows. Below there is example code used for recognizing corrupted data and deleting it:

```
SELECT comments.comments_author_id
     FROM comments
LEFT JOIN USERS
ON user_id = comments.comments_author_id
WHERE user_id is NULL

DELETE FROM comments
WHERE comments.comments_author_id NOT IN(
SELECT USERS.user_id FROM USERS)

DELETE FROM answers
WHERE answers_author_id NOT IN (SELECT USERS.user_id FROM
USERS)
```

On a figure below we can see final result of creating relationships between tables



Problems with SQL database:

- The structure of data is strictly set during the creation of database, therefore is important to deeply analyze the data and set the best structure of tables and relations, because it is difficult and takes a lot of time to change it later.
- With more complex datasets, the relations may become very complicated, so it may be difficult to get an idea of the data.
- It may be hard to look to query for complicated relations and dependencies between data
- Is slower than neo4j

Advantages of using SQL database:

- SQL syntax is very intuitive and is a standardized structured query language approved by ISO and ANSI for relational databases.
- Setting up database is significantly faster than in neo4j
- Easily accessible - there is a lot of different distributions of SQL databases, and most of them have free edition which doesn't have that many limitations

7. Data inaccuracies:

While creating databases we found some inaccuracies in used data:

- Some students in data can also post answers because of data inaccuracy - found with SQL while exploring data,
- Professionals can also have school_membership - found with SQL while exploring data,
- Students and Professionals don't have the same tag names - found with Neo4j while querying both users and tags,
- Some nodes don't have authors - found with SQL while creating relations.