A Graph Database of Yelp Dataset Challenge 2018 and Using Cypher for Basic Statistics and Graph Pattern Exploration

Michael Kronmueller
CECS
University of Louisville
Louisville, KY, USA
mjkron01@louisville.edu

Dar-jen Chang
CECS
University of Louisville
Louisville,KY,USA
djchan01@louisville.edu

Hanqing Hu
CECS
University of Louisville
Louisville,KY,USA
hanqing.hu@louisville.edu

Ahmed Desoky
CECS
University of Louisville
Louisville, KY, USA
ahmed.desoky@louisville.edu

Abstract—**In this paper, we use Neo4j, a popular graph database, to store the Yelp Dataset for 2018 Challenge, which is a real-world dataset. The graph database provides persistent availability for users to retrieve data using Neo4j Graph Query Language called cypher, for many applications. Users can use Neo4j clients such as Python and R together with cypher and server plugins such as APOC and graph algorithm library to explore the dataset. We explain the basic concepts and applications of cypher graph pattern language. To demonstrate how the database can be used, we use cypher to obtain basic statistics of the dataset and use cypher with graph algorithm library to explore interesting graph patterns such as bipartite and connected components.**

Keywords— graph database, Neo4j, bipartite, graph algorithms

I. Introduction

Yelp Dataset 2018 Challenge challenges students to use Yelp data in innovative ways and break ground in research [1]. In this paper we describe a graph database designed to store the dataset and use a graph query language to explore the database. We focus on methods of data exploration rather than on specific analytics results and applications. In section II, we give information about the dataset, while in section III, we describe how we use Neo4j, a popular graph database system, to store the dataset. In section IV, we explain the cypher graph pattern language, which is used to obtain basic statistics and explore bipartite and connected components, as discussed in section V and VI, respectively. In section VII, we describe our future work.

II. YELP DATASET 2018

The Yelp dataset 2018 includes User (i.e. registered members of Yelp), Business (including Category of each Business), Review written by User on Business, Tip given by User for general comment on Business, and others (e.g. Photos of food). In this paper, only User, Business, Review, and Tip data files are considered. Figure 1 show size information of these four data files in JSON format.

File Name	File Size	Number of records
user.json	1,847,071 KB	1,326,101
business.json	141,773 KB	174,567
review.json	4,099,872 KB	5,261,669
tip.json	192,928 KB	1,098,325

Fig. 1 Yelp data files size information

Each record in User includes user_id, name, review_count, average_stars, yelping_since (i.e. when the user registered as a Yelp member), friends (a list of the user's friends), elite (a list of years in which the user is an elite member), etc. Each record in Business includes business_id, name, address, city, state, review_count, stars, categories, etc. Each record in Review includes review_id, user_id (the user who wrote the review), business_id (the business reviewed by the review), stars (5-stars rating of the business), date, useful, funny, cool, and text. The Review fields, useful, funny, and cool, are integer scores to indicate usefulness and sentiment of the review. Each record in Tip includes user_id, business_id, date, likes, and text. Note there is no tip_id given in Tip.

III. A GRAPH DATABASE OF YELP DATASET

In general, there are two approaches to using the dataset, namely, language centric and database centric. The language centric is to use a programming language such as Python and R to import the dataset to memory-resident data structures such as data-frames and use it for a specific application. The database centric is to import the dataset to a database, which users can extract data needed for many different applications. We use database centric to store the dataset.

There are two types of database systems, SQL (i.e. relational database) and NOSQL (Not Only SQL), in use today. We decide to use a graph database, a type of NOSQL, to store the dataset to take advantages of its explicit representation of connected data and its visual graph data model for a more transparent view of the data semantics. Furthermore, a graph data query language, based on path pattern matching, provides a powerful mean to retrieve complex graph patterns. Specifically, we use a popular graph database system called Neo4j [2] [3] to store the dataset. Neo4j graph databases support a graph data model called labeled property graph and a Graph Query Language (GQL) [4] called cypher.

The labeled property graph as used in Neo4j is a graph consisting of labeled Nodes (so that each Node has a distinct label or class, which can be used to describe its functional role and for faster node retrieval) and named Relationships as directed edges connecting nodes. Nodes and Relationships have properties. Although the labeled property graph is required to be directed, Neo4j GQL, cypher, can navigate a path in both directions. As a result, a Neo4j graph can be used as directed or un-directed graph based on its user's commands.

Between two nodes, there can be only one Relationship of a given name. Multiple relationships of different names between nodes, however, are allowed and very common. Cypher is similar to SQL in language structure (both based on clauses), but cypher matches graph patterns, rather than matching text, to retrieve desired information from the graph stored in the database. Our design of the Neo4j graph data model to store the dataset is shown in Figure 2. In the graph database, a node of User label stores a User record from the dataset, where record fields become properties of the node. For example, the name field of a user record is stored as the node property called name, as expected.

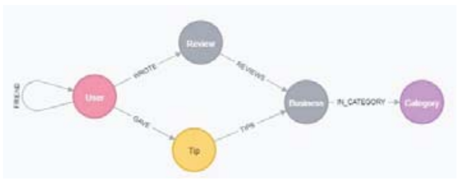

Fig. 2 Graph Data Model for Yelp Dataset

To reference a node property, we use the dot notation. For example, n.name references the name property of the node n. There is a relationship, named WROTE, from User nodes to Review nodes. The relationship is inferred from the user_id field of the Review JSON file. Since this relationship is explicitly represented as a graph edge, there is no need to store user_id in the Review nodes. In a relational database design of the dataset, we may need to store user_id in a column of the Review table and use it as a foreign key to reference the user who wrote the review. Because of this design difference in graph and relational database, to retrieve relationship between two entities, a graph database uses edge traversal, whereas a relational database uses join operation.

We imported the dataset to a Neo4j graph database using the APOC library [5] and scripts taken from Neo4j Graph Algorithm library user guide [6] with some modifications. Once the database is in place, users can use any Neo4j clients, including Neo4j Browser and language drivers for Java, C#, Python, R, etc., to connect and query the database. Cypher is very expressive for doing OLTP (On-Line Transaction Processing), but is limited in doing OLAP (On-Line Analytics Processing). For OLAP, users can use Java to implement user-defined functions and procedures library to extend the functions of Neo4j server, which are similar to how stored procedures are used to extend SQL. APOC and Graph Algorithm libraries are two of such libraries. APOC [5] is designed for general-purpose applications, whereas the Graph Algorithm library [6] is designed for efficient execution of graph algorithms. The supported graph algorithms include shortest paths, page-rank, centrality, connected components, minimum spanning trees, community detection, and so on. To use these libraries all you need to do is call their functions or procedures in cypher statements possibly with subgraphs parameters defined by using the cypher graph pattern language. In the next section, we overview the cypher graph pattern language of sufficient detail with examples so that readers can follow the cypher statements given in the remaining of the paper.

IV. CYPHER GRAPH PATTEN MATCHING

To introduce cypher graph pattern match we start with a simple query against the Yelp database (Fig. 2). Say we want to find the user name of a user who wrote the review of a giving review id. In cypher, it looks like this

```
match (u:User)-[:WROTE]->(r:Review)
where r.id = Someid
return u.name
```

In SQL we use join to connect relationships, whereas in cypher, we define graph pattern to match connected relationships. The previous cypher statement illustrates matching a simple path pattern and using a where clause to filter desired result.

Cypher graph patterns are built from three kinds of graph patterns, namely, node pattern, relationship pattern, and path pattern. The cypher graph pattern language is kind of ASCII Art, which uses text to represent pictures to mimic the shape of node, relationship, and path commonly used to draw graphs on paper. We will define these three types of graph patterns and demonstrate their applications by answering some queries against the Yelp database.

Node pattern

The following examples give the most commonly used node pattern in cypher syntax:

```
()          Match node of any label
(x)         Variable x bound to a node of any label
(:Usr)      Match any User node
(x:User)    Variable x bound to any User node
(x {name:'John'}) Variable x bound to any node given
                    a property value
```

Example Find the business id of up to 10 open businesses in Cleveland city.

```
match (b:Business {city:'Cleveland',is_open:1})
 return b.id limit 10;
```

Relationship Pattern

The following examples give the most commonly used relationship pattern in cypher syntax:

```
-[]->         allow navigating any relationship
              in the defined direction
-->           same as previous relationship pattern
-[:WROTE]->   allow navigating a named relationship
              in the defined direction
-[:WROTE]-    allow navigating a named relationship
               in both directions
-[r:REVIEWS]- variable x bound to the named
               relationship in both directions
-[*2]->        allow navigating in any defined
              relationship direction exactly 2 hops
-[*3..5]->    allow navigating in any defined
              relationship direction at least 3 and
              up to 5 hops
 -[:FRIEND*]-allow navigating in both direction of
             FRIEND relationship any number of hops
```

Notes: (1) relationship properties can be given just like node properties using { and } to enclose a list of properties in key/value format separated by comma. (2) Relationship patterns cannot be used alone to match graph patterns. They are mainly used to form path patterns.

Path pattern

A path pattern is an alternating sequence of node and relationship patterns. The following examples show some commonly used path patterns in cypher syntax:

```
(a)-->(b) (same as (a)-[]->(b))
(a)-[*2]->(b) (same as (a)-->()-->(b))
(a)--(b)<--(c)-->(d)
(u1:User)-[:FRIEND*2]-(u2) (match friends of friends)
(u:User)-[:WROTE]->(:Review)-[:REVIEWS]->(b:Business)
```

The last path pattern says User u wrote a Review on Business b. Since each user wrote at most one Review on a given Business, for each (u, b) pair, none or one path in the database matches the path pattern. The path pattern matches the subgraph containing the matched paths for all (u,b) pairs.

Example Find the user's name up to 5 occurrences of any user who wrote a review on any business in 'Hair Salons'.

```
match (u:User)-->()-->(b:Business)
match (b)-[:IN_CATEGORY]->(c)
where c.id = 'Hair Salons'
return u.name limit 5
```

Example For each user, find the user's name and total number of reviews written by the user. Return the result in descending order of the number of reviews up to 5 users.

```
match (u:User)-[:WROTE]->(r:Review)
with u, count(r) as NumberOfReviews
return u.name, NumberOfReviews
order by NumberOfReviews desc limit 5
```

The with clause simply passes the current result to the next cypher statement for further processing. We need to do this because if we compute the aggregate function, count(), based on the grouping by the user's name, the result may be wrong (multiple users may have the same name).

The total number of reviews written by user u is essentially the out degree of u in the WROTE relationship. Thus we can give a more efficient cypher statement calling the apoc.node.degree function as shown below:

```
match (u:User)
return u.name,
      apoc.node.degree(u,'WROTE>') as NumberOfReviews
order by NumberOfReviews desc limit 5
```

Chapter 3 of the Neo4j Developer Manual [7] gives a more detailed description of cypher. For future evolution and specification of cypher, the openCypher website [8] gives many useful resources.

V. BASIC STATISTICS

Node and relationship count

The following simple cypher queries can be used to get node and relationship counts:

```
match (n) return count(n)
match (u:User) return count(u)
match ()-[r]->() return count(r)
match ()-[r:REVIEWS]->() return count(r)
```

Without Friends and FRIEND relationships, in the database there are

7,861,955 nodes, and
13,387,510 relationships

Number of reviews per user

For number of reviews per user, we can use the following cypher statement with APOC functions:

```
match (u:User)
Return avg(apoc.node.degree(u,'WROTE>')) as Avg,
       max(apoc.node.degree(u,'WROTE>')) as Max,
       min(apoc.node.degree(u,'WROTE>')) as Min,
       stdev(apoc.node.degree(u,'WROTE>')) as Stdv
```

The returned result is listed in Figure 3.

Avg	Max	Min	Stdv
3.9678	3569	1	13.710

Fig. 3 Summary statistics of number of reviews per user

The distribution of number of reviews per user is shown in Figure 4. The distribution is a typical power law distribution commonly seen in degree distribution of social networks. The number of users who wrote more than 30 Reviews only counts for 1.51% of the total number of users. But in terms of review number, still there are 3,411 users who wrote 100 or more reviews and 12 of them wrote 1000 or more reviews.

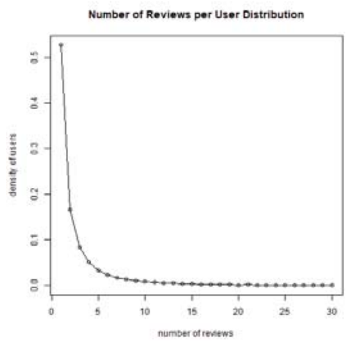

Figure 4 Distribution of number of reviews per user

Only one of the 12 users is not an elite member in any year. It would be interesting to study why this user is so active in voicing his or her opinions.

Business

In the dataset, businesses are located in 1,094 cities, of which majority are in U.S and Canada. There are 1,293 business categories.

Number of reviews per business

The summary statistics of number of reviews per business is given in Figure 5.

Avg	Max	Min	Stdv
30.141	7362	1	98.217

Fig. 5 Summary statistics of number of reviews per business

The number of reviews per business distribution is shown in Figure 6.

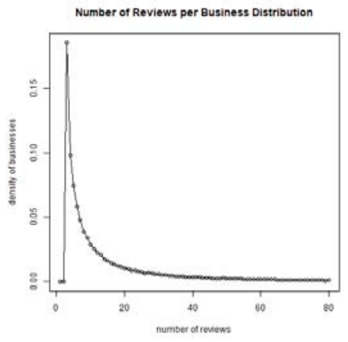

Fig. 6 Distribution of number of reviews per business

The density of the number of reviews per business large than 80 is all below 0.002. The peak on the distribution curve (Fig. 6) is about 0.185 at 3 reviews/business. On the large review/business side, there are 63 businesses of more than 2,000 reviews, 22 of more than 3,000, 3 in 4,000s, 2 in 5,000s, and 2 in 7,000s. It is worthwhile to learn why these businesses have a much larger number of reviews than those of most businesses.

VI. GRAPH PATTERN EXPLORATION

Once the graph database is in place and basic statistics are known, many questions and applications can be asked of the dataset. In this section, we discuss how we explore the graph database for some graph patterns including bipartite projection and connected components, and use them to get some interesting facts about the dataset. The purpose of the discussion is to show how cypher and graph algorithms can be used to explore the graph database in general.

Bipartite projection

A bipartite is a graph whose nodes are divided into two sets X and Y, and only the links between X and Y nodes are allowed. The graph illustrated in Fig. 7 is a bipartite. A bipartite can be either directed or un-directed.

Many real-world networks are bipartite or contain bipartite pattern. For example, in the Yelp graph model (Fig. 2), the subgraph,

```
(:User)-[:WROTE]->(:Review)-[:REVIEWS]->(:Business)
```

is a bipartite pattern between User and Business nodes using Review as a linking relationship between these nodes. Cypher graph pattern language can be used to elucidate bipartite concepts much more concisely than using words. For example, the bipartite in Fig. 7 can be simply expressed in the following graph pattern

```
(:X)-[:LINK]->(:Y)
```

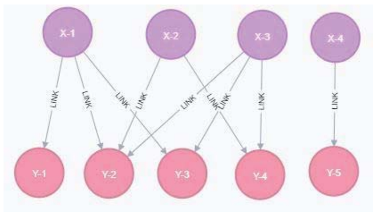

Fig. 7 An example of bipartite

Bipartite projections are commonly used in discovering hidden pattern in either side of bipartite based on the role they play in the bipartite relationship. Bipartite X projection of (:X)-[:LINK]->(:Y) is based on the following graph pattern (x1 and x2 are linked via a common y):

```
(x1:X)-[:LINK]->(y:Y)<-[:LINK]-(x2:X)
```

There are two types of projections, simple and weighted. In simple projections, we just create a linked structure. For example, the simple X projection can be easily expressed in the following cypher statement:

```
match (x1:X)-[:LINK]->(y:Y)<-[:LINK]-(x2:X)
where id(x1) < id(x2)
create (x1)-[px:PX]->(x2)
```

This statement creates a graph consisting of X nodes and relationships, called PX, between X nodes. In many applications, to make use of bipartite projections, however, we need to assign weights to relationships to indicate relative importance of the links. A weighted projection is a simple projection with a weight assigned to each edge. The weight assignment can either be symmetrical or asymmetrical. An asymmetric weight assignment assigns a weight to one direction of an edge and a different weight to the opposite direction of the edge. There are many methods to assign weights in a weighted projection, which all depend on the intended use of the weighted projection. A simple weighted X projection (co-occurrence) is to count the number of linked y common to two X nodes. For example, in scientific collaboration networks [7], the simple co-author weight is the number of papers which have these two authors in their author list. This weighted X projection can be formulated in the following cypher statement:

```
match (x1:X)-[:LINK]->(y:Y)<-[:LINK]-(x2:X)
where id(x1) < id(x2)
with x1, x2, count(y) as ys
create (x1)-[px:PX {weight:ys}]->(x2)
```

The where condition is to make sure x1 and x2 are distinct (so no loops) and make only one link between them (i.e. link x1 to x2 and not x2 to x1). For such a pair, (x1, x2), we count the number of y and set it to ys and create a link from x1 to x2 with a weight set to ys. Note the total number of such (x1, x2) pairs is proportional to N2, where N is the number of X nodes. As a result, computation of bipartite projections generally is a very time consuming task for large graphs.

In the simple weighted projection, each common linked y has equal weight contribution. But in some applications, equal weight contribution may not be suitable. To use co-authorship to measure how much two authors know each other as example, Newman [7] explained "two scientists whose names appear on a paper together with many other coauthors know one another less well on average than two who were the sole authors of a paper." He suggested in such a case each common y node contribution to the weight is one divided by (d – 1), where d is the degree of y in the original bipartite. To apply his idea to set the weight in the previous cypher statement, we simply replace count(y) by the following expression:

```
sum(1.0/(apoc.node.degree(y,'LINK<')-1))
```

For example, in the weighted X projection of Fig. 7 bipartite, the simple weight between X-1 and X-3 is 2 (two common y: Y-2 and Y-3), whereas the weight based on Newman's idea is 1.5 (Y-2 has degree 3 and Y-3 has degree 2).

Another commonly used idea of weigh assignment is to use properties of different x1 and x2 links to a common y to construct the weight between x1 and x2. This weight assignment is not dependent only on network topological structures like counting common y and its degrees. We will illustrate this idea using the Yelp dataset. In the Yelp graph database, we consider the bipartite relationship between User and Business and show some useful User and Business projections. Note in the dataset, a user only wrote at most one review on a given Business. Therefore, the link between a User and a Business in this bipartite relationship is simple i.e. no multi-links. Business projections are based on this pattern:

```
(b1:Business)<--(r1)<--(u)-->(r2)-->(b2:Business)
```

The pattern links two Businesses, b1 and b2, via a common user u who wrote reviews r1 and r2 on them, respectively. How to set the weight between b1 and b2 in the projection will depend on what kind of relationship between Businesses to be captured. Say, if we want to capture co-occurrence of User, we can use simple or Newman weight. In the pattern, the reviews r1 and r2 written by the common User u on Business b1 and b2, respectively, have properties, which can be used to set the link weight between b1 and b2 if needed.

User projections are based on this pattern

```
(u1:User)-[:WROTE]->(r1)->()<-(r2)<-[:WROTE]<-(u2:User)
```

This pattern means for any two Users, u1 and u2, find all paths to link u1 to u2 via a common Business reviewed by reviews r1 and r2 written by u1 and u2, respectively. The common Business in the pattern is anonymous because we only need to know the reviews written by u1 and u2 and use the properties of these reviews to set the weight. For example, we like to compare similarity between Users based on their reviews on Businesses. For a given User pair, u1 and u2, in the pattern, we collect all reviews r1 and r2 in the paths and construct two star-ratings vectors from the stars property of r1 and r2, respectively. Then we use any normalized similarity metric (i.e. ranging from 1 to 0, the larger the more similar) of these two real number star-rating vectors to set the weight. For example, this APOC function

```
apoc.algo.cosineSimilarity(vector1, vector2)
```

returns cosine of the angle between two vectors, which is normalized if both vectors are non-negative.

There are many other weight assignment methods in use. In their paper [10], Ying Fan el at. used weighted community modularity to study the effect of weight on community structure of networks and found that weights do have a big influence on community structure, especially on dense networks, based on their investigation of some real-world and ad hoc networks. An interesting weighted projection of asymmetric weights based on resource allocation is discussed in [11] and this weighting method is often used in personal recommendation systems.

Connected components

Using the (Business, User) bipartite, we demonstrate how to use cypher and graph algorithm library to explore the graph patterns of connected component and partition and from the patterns, observe some interesting facts about the dataset. We focus on the methodology rather the interpretation of the results.

Sine computing business projection is very time-consuming for large number of businesses, as an example, we will consider only all businesses in the city of Cleveland. As such, the total number of these businesses is 3,322 and there are 92,314 reviews written on them by 35,548 users. We can compute the simple weighted Cleveland business projection using the following cypher statement:

```
match (b1:Business)
where b1.city="Cleveland"
match (b2:Business)
where b2.city="Cleveland" and id(b1) < id(b2)
match (b1)<--(r1)<--(u)-->(r2)-->(b2)
with b1, b2, count(*) as coo where coo >= 1
create (b1)-[bp:BP {weight:coo}]->(b2)
```

The pattern, (b1)<--(r1)<--(u)-->(r2)-->(b2), used in the cypher statement is compact and easy to understand, but it is hard for cypher to find an execution plan efficient in memory and time. In real cypher applications, execution efficiency is as important as clarity of query formulation. An equivalent pattern is used latter, which makes a huge difference in execution efficiency. We can compute the connected components of the projection graph using the following graph algorithm library procedure:

```
CALL algo.unionFind('Business', 'BP',
{ partitionProperty:"setId", weightProperty:'weight'})
YIELD nodes, setCount
```

In the result returned from this procedure, the largest component contains 3,218 businesses and each of the other 104 components contains just one business. We say two businesses have a co-reviewed relationship if there is a user who wrote two reviews, one for each of these two businesses. The connected component result means 104 Cleveland businesses do not involve in any co-reviewed relationship. The pattern of having a relatively large component and many small components seems to be common in projections of Business (or User) for different values of coo (co-occurrence) threshold. For example, for coo >= 4, the projection graph has one large component containing 1,218 businesses, one component containing 4 businesses, and 2,100 components each of which contains one business. As coo threshold increases, the size of the largest component in the projection graph decreases.

In the Business projection graph, the degree of a Business node is the number of businesses with which the business has a co-reviewed relationship. We observe there is a business which has the largest number (2,128) of co-reviewed relationships. The weight of an edge connecting two business nodes is the total number of co-reviewers between them. We find out the largest number of co-reviewers is 125 and only the pair of these two businesses, "West Side Market" and "Great Lakes Brewing Company", has the largest co-reviewers.

We run the simple weighted projection with coo threshold set to 10. The resulting projection graph has a component containing 462 businesses and 2,860 components, each of which contains one business.

To create a similarity weighted Businesses projection, we can use the following cypher statement:

```
match (b1:Business)
where b1.city="Cleveland"
match (b2:Business)
where b2.city="Cleveland" and id(b1) < id(b2)

match (b1)<-[:REVIEWS]-(r1:Review)
match (r1)<-[:WROTE]-(u:User)
match (b2)<-[:REVIEWS]-(r2:Review)
match (r2)<--(u)

with b1, b2, count(*) as coo, collect(r1.stars)
 as s1, collect(r2.stars) as s2 where coo >= 5
with b1, b2,
     apoc.algo.cosineSimilarity(s1,s2) as s
where s > 0
create (b1)-[bp2:BP2{weight:s}]->(b2)
```

Note in this cypher statement, we break the bipartite pattern, (b1)<--(r1)<--(u)-->(r2)-->(b2), into four match patterns, which lead cypher to construct a much more efficient execution plan for matching the bipartite pattern.

VII. FUTURE WORK

For future work we plan to pre-process Review texts, for example, retrieving bags of words and analyzing text sentiments. The results of this text processing can be inserted into the graph database linked to the Review nodes. Once the links are made, text results are connected to all the other graph components such as User and Business nodes. This connected text information opens up many possible applications. For example, we can study the general review text writing style of a user and use it to detect style change of a new review written by the user for spam prediction.

REFERENCES

[1] Yelp Dataset Challenge https://www.yelp.com/dataset/challenge

[2] Neo4j https://neo4j.com/

[3] Ian Robinson, Jim Webber & Emil Eifrem, Graph Databases, Second Edtion, O'Reilly Media, Inc., 2015.

[4] Renzo Angles, el at, Foundations of Modern Query Languages for Graph Databases, ACM Computing Surveys, Volume 50 Issue 5, November 2017.

[5] APOC User Guide https://neo4j-contrib.github.io/neo4j-apoc-procedures/

[6] Neo4j Graph Algorithms User Guide https://neo4j-contrib.github.io/neo4j-graph-algorithms/

[7] The Neo4j Developer Manual https://neo4j.com/docs/developer-manual/current/

[8] openCypher https://www.opencypher.org/

[9] M. E. J. Newman, Scientific collaboration networks. II. Shortest paths, weighted networks, and centrality, PHYSICAL REVIEW E, VOLUME 64, 016132, 2002

[10] Ying Fan, el at., The effect of weight on community structure of networks, Physica A 378, 2007, 583–590

[11] Tao Zhou, el at., Bipartite network projection and personal recommendation, PHYSICAL REVIEW E 76, 046115, 2007