IJoL International Journal of Librarianship

ISSN: 2474-3542 Journal homepage: http://journal.calaijol.org

An Agglomerative-adapted Partition Approach for Large-scale Graphs

Tao Chen, Rongrong Shan, Hui Li, Dongsheng Wang and Wei Liu

Abstract:

In recent years, an increasing number of knowledge bases have been built using linked data, thus datasets have grown substantially. It is neither reasonable to store a large amount of triple data in a single graph, nor appropriate to store RDF in named graphs by class URIs, because many joins can cause performance problems between graphs. This paper presents an agglomerative-adapted approach for large-scale graphs, which is also a bottom-up merging process. The proposed algorithm can partition triples data in three levels: blank nodes, associated nodes, and inference nodes. Regarding blank nodes and classes/nodes involved in reasoning rules, it is better to store with an optimal neighbor node in the same partition instead of splitting into separate partitions. The process of merging associated nodes, needs to start with the node in the smallest cost and then repeat it until the final number of partitions is met. Finally, the feasibility and rationality of the merging algorithm are analyzed in detail through bibliographic cases. In summary, the partitioning methods proposed in this paper can be applied in distributed storage, data retrieval, data export, and semantic reasoning of large-scale triples graphs. In the future, we will research the automation setting of the number of partitions with machine learning algorithms.

To cite this article:

Chen, T., & et al. (2019). An Agglomerative-adapted Partition Approach for Large-scale Graphs. *International Journal of Librarianship, 4*(1), 3-18.

INTERNATIONAL JOURNAL OF LIBRARIANSHIP, 4(1), 3-18
ISSN: 2474-3542

An Agglomerative-adapted Partition Approach for Large-scale Graphs

Tao Chen, Shanghai Library/Institute of Scientific & Technical Information of Shanghai, Shanghai, China; School of Information Management, Nanjing University, Nanjing, China

Rongrong Shan, Department of Library, Information & Archives Shanghai University, Shanghai, China

Hui Li, Shanghai Library/Institute of Scientific & Technical Information of Shanghai, Shanghai, China; School of Information Management, Nanjing University, Nanjing, China

Dongsheng Wang, Computer Science, University of Copenhagen, Copenhagen, Denmark

Wei Liu, Shanghai Library/Institute of Scientific & Technical Information of Shanghai, Shanghai, China

ABSTRACT

In recent years, an increasing number of knowledge bases have been built using linked data, thus datasets have grown substantially. It is neither reasonable to store a large amount of triple data in a single graph, nor appropriate to store RDF in named graphs by class URIs, because many joins can cause performance problems between graphs. This paper presents an agglomerative-adapted approach for large-scale graphs, which is also a bottom-up merging process. The proposed algorithm can partition triples data in three levels: blank nodes, associated nodes, and inference nodes. Regarding blank nodes and classes/nodes involved in reasoning rules, it is better to store with an optimal neighbor node in the same partition instead of splitting into separate partitions. The process of merging associated nodes, needs to start with the node in the smallest cost and then repeat it until the final number of partitions is met. Finally, the feasibility and rationality of the merging algorithm are analyzed in detail through bibliographic cases. In summary, the partitioning methods proposed in this paper can be applied in distributed storage, data retrieval, data export, and semantic reasoning of large-scale triples graphs. In the future, we will research the automation setting of the number of partitions with machine learning algorithms.

Keywords: linked data, agglomerative-adapted partition, merging algorithm, large-scale graph, *k*-graph

INTRODUCTION

With the rapid development of linked data, more and more organizations are using this mature technology to build and publish their knowledge bases or datasets (Erkimbaev, Zitserman, Kobzev, Serebrjakov and Teymurazov, 2013; Knoblock et al, 2017; Chen Tao, Zhang Yongjuan,

Liu Wei and Zhu Qinghua, 2019). As can be seen from the latest linked open data (LOD) cloud[1], there is a growing number of big datasets, such as DBpedia[2], SciGraph[3], VIAF[4], UniProt[5], and so on. All of these large datasets have become the infrastructure and core components of their fields. For example, the 2016-04 release of the DBpedia dataset describes 6.0 million entities, consisting of 9.5 billion RDF triples. DBpedia data is categorized into hundreds of entity classes, and has been linked by a number of applications and datasets. These large datasets often provide segmented downloads for different classes in official publishing sites. When these databases can be applied, we need to dump and restore them in a local repository which also introduces a number of challenges.

The Resource Description Framework[6] (RDF) is a family of World Wide Web Consortium (W3C) specifications originally designed as a metadata data model. It has come to be used as a general method for conceptual description or modeling of information that is implemented in web resources. Considering the large amount of data, it can lead to performance problems, especially restoring these data in a graph. In fact, normally we only need to operate data on its sub-graph, such as retrieval, export, reasoning in some classes. Some researchers might think of splitting the graph into several named graphs, which is having multiple RDF graphs in a single document/repository and naming them with URIs. If we divide them on average, the data for the one class can be divided into multiple graphs at random. If we create named graphs by class URIs, a larger number of graphs are generated. Excessive joins in federated queries can also cause performance degradation. Therefore, this is not the better solution, especially when the data size between classes is unbalanced. For example, some named graphs only have hundreds of triples, while some others have millions of triples. In view of this background, we propose a simple agglomerative-adapted partition approach that can be used to split triples in large-scale graphs.

LITERATURE REVIEW

A great deal of research articles and presentations have addressed the topic of triples partitions and RDF performance in application. These studies mainly focus on the use of distributed framework, efficient storage and indexing, and dynamic partitioning algorithms.

Many research efforts have been devoted to develop distributed RDF data management systems implemented on the Hadoop computing framework and MapReduce algorithm. For example, Apache Jena Elephas[7] is a set of libraries which provide various basic building blocks for writing Apache Hadoop based applications which work with RDF data. Some researchers use Hadoop to store and retrieve large numbers of RDF triples stored in flat text files in Hadoop Distribute File System (HDFS) (Mohammad, Pankil, Latifur and Bhavani, 2009; Kurt and Richard, 2010; Khushboo and Abhishek, 2017). Vaibhav, Murat, Bhavani and Paolo (2012) proposed Jena-HBase, which stored RDF indices in HBase and directly carried out queries

[1]https://lod-cloud.net/
[2]https://wiki.dbpedia.org/
[3]https://scigraph.springernature.com/
[4]https://viaf.org/
[5]https://www.uniprot.org/
[6]https://www.w3.org/RDF/
[7]https://jena.apache.org/documentation/hadoop/

through HBase APIs. Nikolaos, Ioannis, Dimitrios and Nectarios (2012) demonstrated the *H2RDF* system, a distributed RDF store that combines a multiple-indexing scheme with BigTable and MapReduce. Alfredo, Rajkumar, Vincenzo and Giovanni (2017) provided a MapReduce-model-based algorithm for managing big RDF graphs, which tried to exploit the computational power offered by the MapReduce processing model.

There are also a number of researchers who have made breakthroughs in data storage and indexing. Zeng, Yang, Wang, Shao and Wang (2013) introduced Trinity.RDF, a distributed, memory-based graph engine for web scale RDF data stored in its native graph form. It achieved much better performance for SPARQL queries than the state-of-the-art approaches. Gu, Hu and Huang (2014) proposed Rainbow, a scalable and efficient RDF triple store. The RDF data indexing scheme in Rainbow is a hybrid one which was designed based on the statistical analysis of user query space. Li and Heflin (2010) presented a query optimization algorithm to identify the potentially relevant Semantic Web data sources using structural query features and a term index. Razen, Yasser and Panos (2014) presented PHD-Store, a SPARQL engine for federations of many independent RDF repositories. PHD-Store followed an adaptive approach that allowed it to start processing queries immediately, thus minimizing the data-to-query time, while distributed and indexed only those parts of the graph that benefit the most frequent query patterns.

Some other researchers use graph partitioning optimization techniques about RDF graph patter matching. For instance, Ruben, Miel and Pieter (2014) introduced Linked Data Fragments (LDF), a publishing method that allowed efficient offloading of query execution from servers to clients through a lightweight partitioning strategy. Huang and Daniel (2016) introduced a dynamic graph partitioning algorithm, designed for large, constantly changing graphs, and he also proposed a partitioning framework that adjusted on the fly as the graph structure changed. Wang and Kenneth (2012) proposed a promising approach that utilized the graph nature of RDF datasets to minimize relations among partitions after dataset partitioning, and optimized system design based on it. Hao, Li, Yuan and Jin (2017) described an association-oriented streaming graph partitioning method named Assc. This approach first computed the rank values of vertices with a hybrid approximate PageRank algorithm, and then split these vertices with an adapted variant affinity propagation algorithm.

The agglomerative-adapted approach proposed in this paper belongs to the third category mentioned above. This approach is mainly based on the ontology structure and RDF data, and merges the different nodes from the aspect of measurement.

METHODS

Building the reformation graph

Several key concepts need to be pointed out before the key approaches and methods are introduced. An ontology is an explicit specification of a conceptualization which describes individuals (instances), classes (concepts), attributes/properties, and relations. In knowledge graphs, it describes classes with nodes and uses directional edges to represent attributes. As shown in Figure 1, nodes c1, c2, c3, ... are Classes, and p1, p2, p3, ... represent properties. There are two type of properties, DatatypeProperty and ObjectProperty that describe what kind of values a triple with the property should have. Datatype properties relate individuals to literal data

(e.g. strings, numbers, datetypes, etc.) like p1, p2, p3 in Figure 1, whereas object properties relate individuals to other individuals like properties p5, p6, p7, etc.

In Figure 1, there are also some different types of nodes/classes: standalone node, association node, blank node, and inference node.

A standalone node is a node without relationships with other nodes. This type of nodes only has datatype properties, such as node c1 in Figure 1 has p1, p2 and p3 properties.

An associated node, most commonly used in knowledge graphs, has an association with another node through object property. For instance, node c2 and node c5 are associated with property p5; node c6 and node c9 are linked with property p10.

A blank node is a node in an RDF graph that represents a resource (anonymous resource) for which a URI or literal is not given. Blank nodes are recommended to be used with related entity resources as there are different node IDs for the same triples of blank node in two graphs. The node connected to p23 belongs to this node type.

An inference node is a node reasoned by other classes and properties based on inference rules. For example, the node r12 is inferred from p16 and p17 properties which belong to node c9 and node c10 respectively.

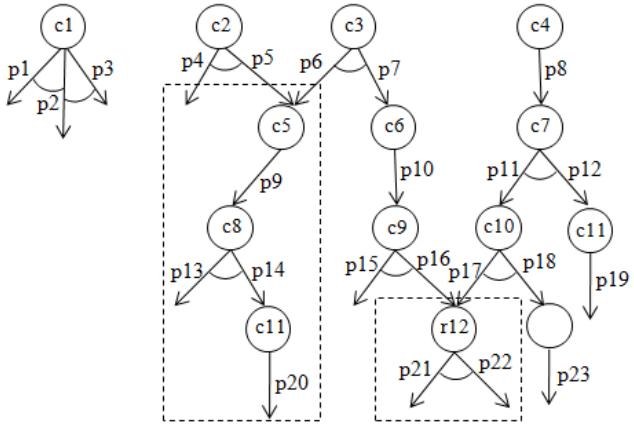

Figure1. A pseudo-reformation path-goal graph constructed

In splitting the reorganization diagram, we can try to get the largest possible trees. As demonstrated in Figure 1, there are four top nodes c1, c2, c3 and c4, thus the easiest way to do this is to split this graph into four trees which is shown in Figure 2. However, if you want to split this graph into more or fewer trees, you can use the method we proposed later.

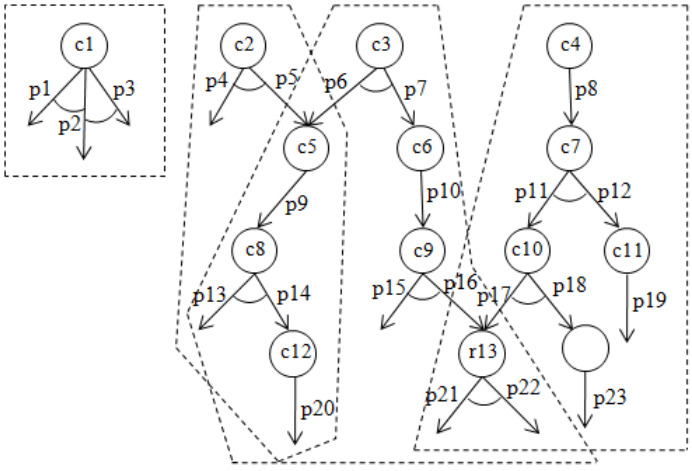

Figure 2. The split trees and the duplicate areas

The significant features of the graph might be the shared nodes such as c5 belongs to c2 and c3 connects with p5 and p6. There are two methods for how these common nodes are partitioned. As many partitions will bring more joins between different partitions, splitting these nodes into partitions separately will cause efficiency problems while making a SPARQL query. However, storing them in multiple partitions can result in duplication of triples. Therefore, we need to make a balance between these two methods, since our original intent is to minimize duplication of data and minimize the number of joins with each other.

Partition process for trees

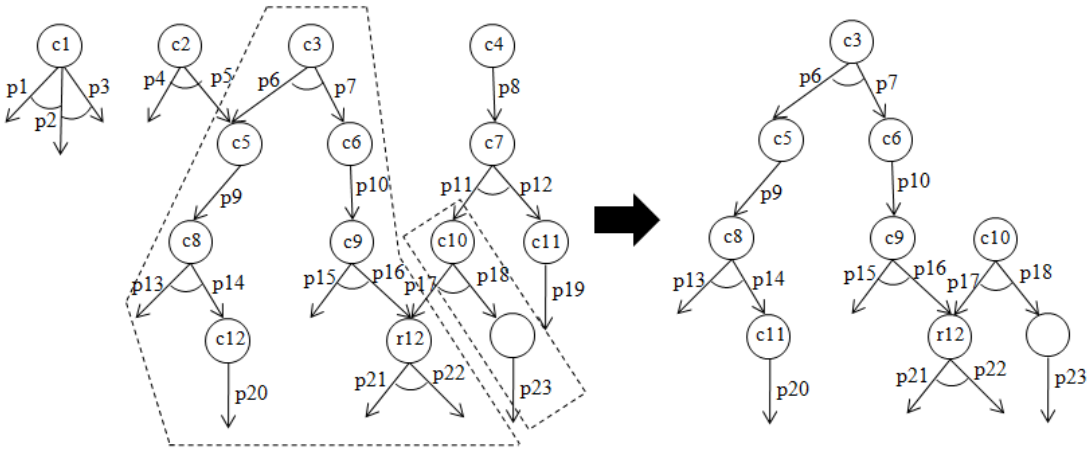

Figure 3. Take the c3-oriented tree from last graph as an example

Given a tree as shown on the right part of Figure 3, we need to extract "partition templates" from the tree. We propose to partition it with an agglomerative-adapted merging process to obtain reasonable sized partitions. Firstly, we define the *cost* of a partition template as the total number of no-duplicate triples that are mapped to their predicates, and define the neighbors of a certain partition as the partitions that share the same nodes/predicates and are next to each other. The basic approach is to start with as many small partitions as possible and always merge the lowest cost partition with its smallest neighbor partition into a new partition. The new partition size and sequential partition list are then updated and continuously moved to the next smallest partition to iteratively partition.

As illustrated in step 0) in Figure 4, we initialize the cost of each predicate by loading the triples and mapping their predicates, and the cost of each minimum partition is simply the sum of the cost of all its predicates. Then we can divide all the nodes into separate partitions, except for blank nodes, which must be put together with their corresponding resources. There are eight initial partitions in this step, and the cost in node c10 and its blank node is 50+115+20=185, of which 50 and 115 are the triple number of p17 and p18 in node c10, and 20 is the triple number of p23 in blank node. Next we have to do four merging steps.Of course, the number of merging times is not fixed. That needs to be based on the final number of partitions to determine.

In step 1), we determine the least partition cost is 35 which is the sum of the quantities of property p13 and p14 of node c8, and we merge it with its neighbor node c5. For node c8, it has two neighbor partitions, c5- and c11- oriented partitions; however, the cost of c5 is 60 fewer than 80 of c11. So we chose the least one, c5, as the best adjacent merge node and the cost of new

partition merged with c5 and c8 is 60+20+15=95. It is important to note that the inference node r12 is not processed at this step. It will be merged in the last step.

Subsequently, in step 2), we continue to search and get the next least partition, which is c6-oriented partition, with cost of 75, and we merge it with its best neighbor partition with least cost. After merging, the new partition includes c3 and c6 nodes, and the cost is 70+20+75=165. Similarly, we repeat this process in step 3) to merge nodes c5, c8, and c11.

The last step is how to merge the inference node, which is most important step in the entire process. The nodes corresponding to the properties used in the inference rules need to be stored in the same partition as the inference node. In step 4), for instance, we should put nodes c9, c10, and r12 in one partition, and the total cost in this partition is 230+185+50=465.

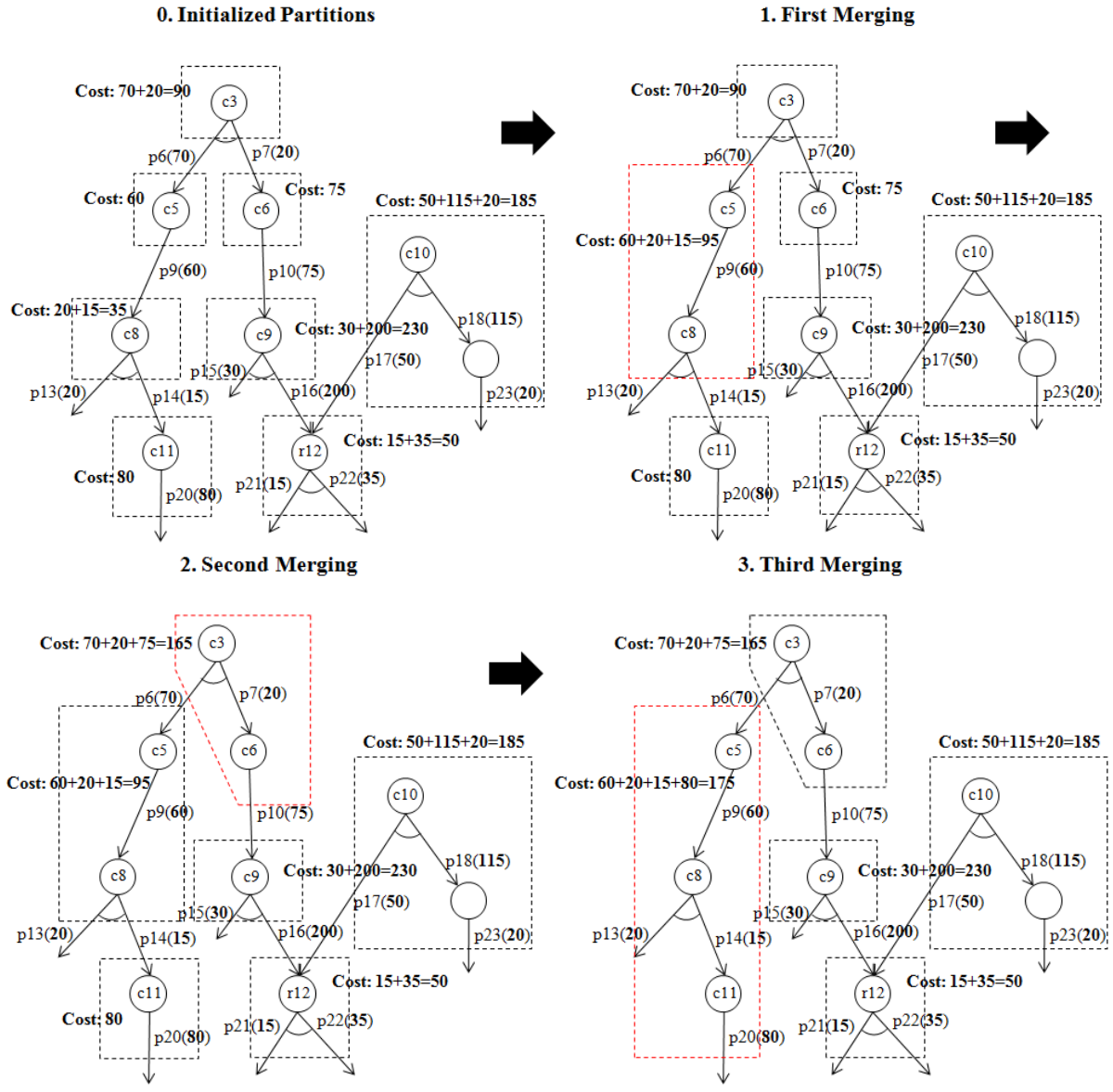

Figure 4. An example of agglomerative-inspired merging process

If we want to split the graph into three partitions, when we merge four times, it has already produced relatively balanced partition sizes, as shown in Figure 5, with costs of 165, 175 and 465. We can further merge partition one (with nodes c3 and c6) and two (with nodes c5, c8 and c11) if we want to split the graph into two partitions. Eventually we can get two partitions with costs of 340 and 465.

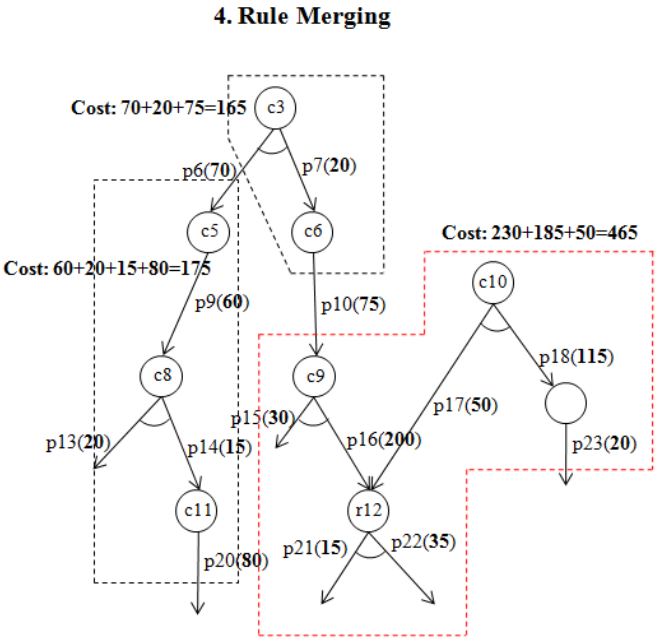

Figure 5. Rule merging process for last example in figure 6

The proposed agglomerative-adapted approach is a bottom-up consolidation process. Note that "bottom" here means the node with the lowest cost and not the leaf of the trees. There are some details and observations about partition principles or rules:

1) Do not split triples belonging to the same resource, but you can split sub-triples that exist in the object resource to another partition.
2) If triples t and t' combined with some axioms (strong relationships) will infer some new triples, they must appear in the same partition.
3) Try to reduce duplicate nodes in partition templates as much as possible.
4) Try to balance the overall sizes of partition templates, avoiding some template sizes getting too large while others too small.
5) A blank node is a resource without global identity, and it must appear in the same partition with its related subject resource.
6) Only split triples corresponding to the object properties, do not split triples of datatype properties.

Traditionally, agglomeration is a hierarchical clustering method that starts to treat each object as singleton clusters and then successively merge pairs of clusters until there is one single cluster that contains all objects. The significant features that could be referred to are 1) the cluster number is flexible and could be obtained by cutting an endrogram and 2) the merging process is to always merge the closest or most similar two clusters.

Note that the situation and application methods are of great difference in our case, but we adapt the ideas of agglomeration to our own partition approach. In this example, we assign very small cost values to simplify the description of the idea. However, the actual costs should be larger, which can produce a Cartesian product.

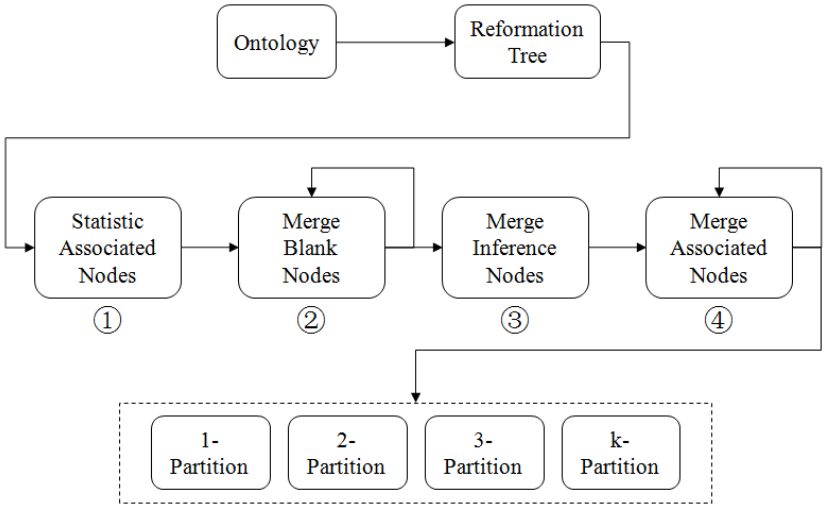

Figure 6. Algorithm flowchart

ALGORITHM AND IMPLEMENTATION

The main flow of the algorithm is shown in Figure 6, where we take four merging steps, of which steps 2 to 4 require iteration to achieve the final desired result. The algorithm should operate the ontology and RDF data, and we give the core SPARQL statements and algorithm with pseudocode of the merging approach in the following.

Step1: Statistic Cost of Associated Nodes

```
1: q ← SELECT ?node (COUNT(DISTINCT *) AS ?cost)
     WHERE {
        ?uri ?p ?o .
        {SELECT DISTINCT ?node ?uri
         WHERE {
            ?uri a ?node .
            FILTER (isIRI(?uri))
         }}
     } GROUP BY ?node
2: entityMaps<node, cost> <-perform this SPARQL query and save the output in entityMap variable.
```

First, we perform the statistics algorithm of associated nodes in Step 1 in the whole raw graph or model (*raw_model*). It is important to note that blank nodes are not counted here, because these nodes will be merged into the corresponding nodes later. After execution, this

algorithm will output all the associated nodes and their cost value, which can be recorded in variable *entityMaps<node, cost>*.

Step2: Merge Costs of Blank Nodes to Associated Node

```
1: void mergeBlankNodes(entityMaps<node, cost>, raw_model) {
2:    while time < entityMaps.size
3:       q ← SELECT DISTINCT ?p
4:            WHERE {
5:              ?uri a <node> ; ?p ?s .
6:              FILTER (isBlank(?s))
7:            }
8:         bp_list<property> <-launch this SPARQL to query all properties which have blank nodes as object value.
9:       IF bp_list.size > 0 THEN
10:          while time < bp_list.size
11:             mergeBlankNode(cost, raw_model, node, "<" + property + ">");
12:          end
13:   end
14: }

15: void mergeBlankNode(cost, raw_model, node, path) {
16:    q ← SELECT (COUNT(DISTINCT *) AS ?count)
17:       WHERE {
18:         ?s ?p ?o .
19:         {SELECT ?s WHERE {
20:            ?uri a <node> ; path ?s .
21:            ?s ?p ?o .
22:         }}
23:       }
24:     cost ← cost + count <-run SPARQL query to count cost of blank node (?s) in property path of associated node.

25:    q ← SELECT DISTINCT ?p
26:       WHERE {
27:         ?uri a <node> ; path ?s .
28:         ?s ?p ?o .
29:         FILTER (isBlank(?o))
30:       }
31:     bp_list<property> <-launch this SPARQL to get all properties which have blank nodes of path resource (?s).
32:    IF bp_list.size > 0 THEN
33:       while time < bp_list.size
34:          path = path + "/<" + property + ">";
35:          mergeBlankNode(cost, raw_model, node, path);
36:       end
37: }
```

Subsequently, with the output of the last step, *entityMaps<node, cost>*, we can use algorithm 2 to merge blank nodes to each associated node. Blank nodes can have multiple layers when used; therefore, this step can be looped and recursively executed, which is shown in function **mergeBlankNode***(cost, raw_model, node, path)*. This algorithm will count the cost of

all blank nodes of each associated node, and the cost will be summarized with the value of the associated node counted from algorithm 1 in line 24. Until now, we have merged blank nodes, and the output can be put into the *entityMaps<node, cost>* variable for subsequent operation. At this point, the nodes in *entityMaps* have not changed, but the costs of blank nodes are summarized.

Step3: Merge Inference Nodes

In a semantic web application, an inference engine or reasoner is used to derive additional RDF assertions, which are entailed from some base RDF together with any optional ontology information, and the axioms and rules associated with the reasoner. All classes involved in inference rules are divided the same partition, which can guarantee the efficiency of reasoning. When merging, the node with less cost will be merged into a large one, and then update *entityMaps* value. The operation of this step is relatively simple and can be only performed in the *entityMaps* variable. After this merging step, the size of *entityMap*s will be changed. The inference nodes will less cost value will be merged into the nodes with more cost.

Step4: Merge Associated Nodes

In this step, we will extract the node with the smallest cost and use the ontology structure to calculate the optimal adjacent node. This is not a simple use of SPARQL to directly query the neighbor nodes, so we can use the following flowchart in Figure 7 to illustrate the algorithm.

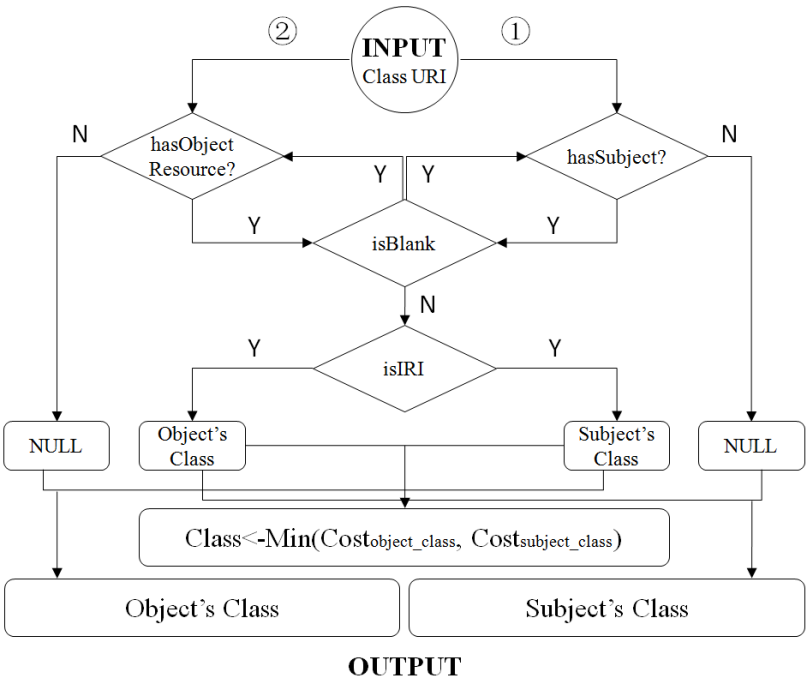

Figure 7. Flowchart for getting neighbor node

As shown in the Figure 7, the input of the algorithm is class or node URI which has the smallest cost value in *entityMaps*, and the output result is the optimal neighbor node, the class of the upper subject or lower object resource of this node. For a node URI, it may be a subject to connect other resources, or it may act as object of other resources. Therefore, it is necessary to find the subject and object classes for this node URI. Because process 1 and 2 are similar, we

will only explain process 1 for looking for an adjacent subject class. First, we need to determine whether the node URI is an object of another resource, that is, to identify a possible subject resource. If such a subject resource does not exist, this indicates that the input node is not connected to another subject resource. Conversely, when such a subject resource exists, the next step is to determine the resource type of this subject, IRI or a blank node. If the subject type is a blank node, it means that this blank node must exist as an object for another resource. We should repeat this step until the end subject of the IRI type is found. Once found, it is convenient to find out the class type with property rdf:type of this resource which is marked subject_class. Based on the same principle, we can find the class of the adjacent object resource marked object_class here. After getting the output result, we select the class with minimum cost in subject_class and object_class as the output value. Of course, if one of subject_class or object_class does not exist, you can use the other class as output value. Finally, the cost of the input class can be merged into the output class, which will update the value of the output class in *entityMaps*, while removing the input class too.

The following pseudocode shows the algorithm implementation of this step in Apache Jena framework. The final result of the merging step recursion is that the size of *entityMaps* is *k*.

```
1: void mergeAssociatedNodes(entityMaps<node, cost>, raw_model) {
2:     entityMap<minClass, min> <-get the node with smallest cost in entityMaps.
3:     relClasses <-define the list of relation classes of node with smallest cost in entityMaps.
4:     mergeUps(relClasses, raw_model, minClass, ""); <-get the subject's class with function mergeUps.
5:     mergeSubs(relClasses, raw_model, minClass, ""); <-get the object's class with function mergeSubs.
6:     mergeNode2RelClass(entityMaps<node, cost>, minClass, relClasses); <-merge minClass node to
the optimal neighbor node.
7:     IF entityMaps.size > k THEN
8:         mergeAssociatedNodes(entityMaps<node, cost>, raw_model); <-after each recursion, the size of
entityMaps will gradually tend to the set value of k.
9: }
10: void mergeUps(relClasses, raw_model, node, property) {<-only offers the program logic of
mergeUps function in paper.
11:     IF property is blank THEN
12:         property ← a
13:     ELSE
14:         property ← property + "/a"

15:     q ← SELECT DISTINCT ?class ?p
16:         WHERE {
17:           ?s a ?class ; ?p ?o .
18:           ?o property <node> .
19:           FILTER (isIRI(?s))
20:         }
21:     IF results.size > 0 THEN <-results are the outputs in SPARQL query.
22:         relClasses.add(class); <-add the class queried in SPARQL to relClasses list.
23:     ELSE
24:         q ← SELECT DISTINCT ?class ?p
25:             WHERE {
26:               ?s a ?class ; ?p ?o .
27:               ?o property <node> .
28:               FILTER (isBlank(?s))
29:             }
```

```
30:     property ← "<" + p + ">/" + property;
31:     mergeUps(relClasses, raw_model, node, property); <-recursive queries until the IRI entity node.
32: }
```

EXPERIMENTS AND ANALYSIS

Figure 8 is a practical case of the bibliographic platform in our actual project. We use the ideas and algorithms we proposed to split the whole graph to partitions with different degrees. The bibliographic platform is developed based on BIBFRAME (Bibliographic Framework), which is a data model for bibliographic description. BIBFRAME was designed to replace the MARC standards and to use linked data principles to make bibliographic data more useful both within and outside the library community.

The BIBFRAME vocabulary consists of RDF classes and properties. Classes include the three core classes, Work, Instance, and Item. Properties describe characteristics of the resource and relationships among resources. The highest level of abstraction, a Work, reflects the conceptual essence of the cataloged resource. A work may have one or more individual which reflects information such as publisher, place, and date of publication. An item is an actual copy (physical or electronic) of an instance. It reflects information such as its location (physical or virtual), shelf mark, and barcode. The three core classes are connected by properties. For example, an Instance may be an "instance of" a particular BIBFRAME Work, and an Item is connected to an Instance via bf:itemOf property.

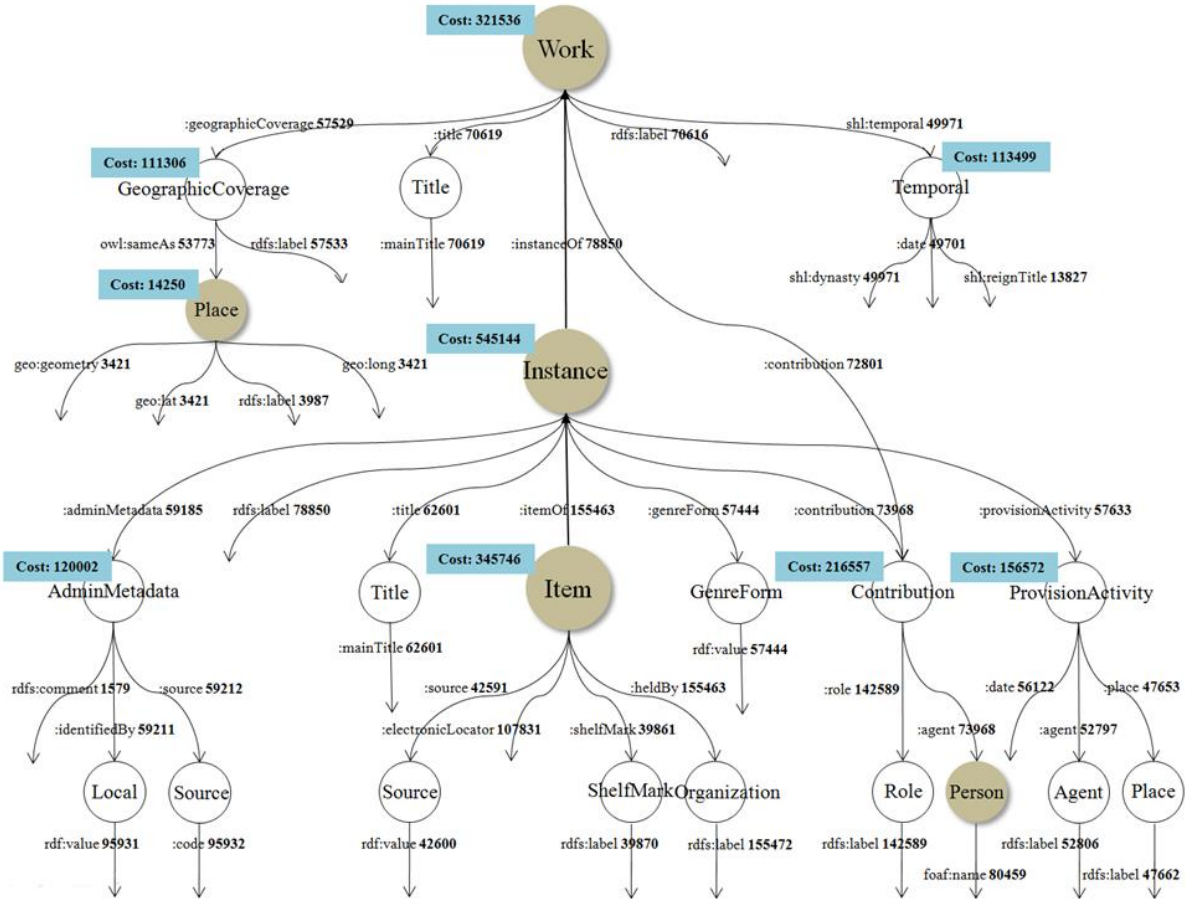

Figure 8. The reformation tree for Bibliographic platform

In this figure, we mark the cost of classes that need to be summarized by several properties. However, there are some classes without a mark that only have one property, and in this case, the cost of class is the same as its property cost. When actually in use, we use a large number of blank nodes without the background color in Figure 8, such as GeographicCoverage, Title, Temporal nodes. The associated nodes, e.g. Work, Instance, Item, Place and Person nodes, are drawn with background color.

In order to verify the feasibility of the proposed algorithm, we consider dividing the entire graph into 5, 4, and 3 partitions, respectively. As seen in Figure 9, the bibliographic platform contains five associated nodes, so we only need to merge blank nodes if we want to split the graph to 5 partitions. When you need to split them into 4 partitions, you must merge the triples data from the Place class into the Work class. Similarly, for 3-graph, the Person class should be merged into the upper classes, Work and Instance. You will find that Person class has been stored in two copies. The advantage of two copies is that the relative independence and integrity of the data can be guaranteed when using partitioned data, which is conducive to the improvement of the efficiency of use. In this project, we didn't use reasoning, so we did not provide the result of inference nodes merging.

In statistics, standard deviation (SD) is a measure that is used to quantify the amount of variation or dispersion of a set of data values. A low SD indicates that the data points tend to be close to the mean of the set, while a high SD indicates that the data points are spread out over a wider range of values. The SD can be calculated with following formula:

$$\mathrm{SD} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

where $\{x_1, x_2, \ldots, x_N\}$ are the observed values of the sample items, μ is the mean value of these observations, and *N* is the number of observations in the sample.

In the SD formula, for this example, the *N* is partitions number, and x_i are costs of nodes.

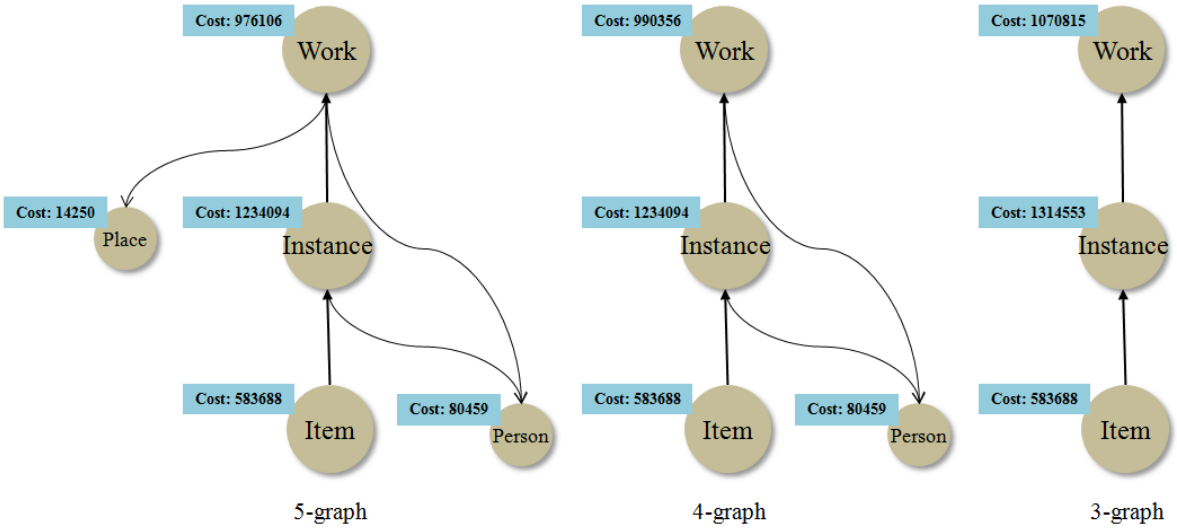

Figure 9. Partitions comparison of different degrees for Bibliographic platform

As we calculated 5-graph, the SD value is 5.3720e+05. With the same method, the SD values are 5.0496e+05 and 3.7213e+05 for 4-graph and 3-graph respectively. As you can see from the results, 3-graph has a smaller SD value.

CONCLUSION AND FUTURE WORK

This paper presents an agglomerative-adapted partition approach for large-scale RDF graphs. According to the characteristics of ontology structure and triples, we propose a bottom-up and multi-layer node-merging algorithm which contains blank nodes merging, associated nodes merging, and inference nodes merging. Relative independence and moderate equilibrium are the main points of consideration in the process of partition segmentation. When merging, blank nodes cannot be split into new partitions, and Classes involved in the inference rules are recommended to be in one same partition. The merging algorithm can be looped based on the desired number of partitions, *k*-graph, and the triples in each partition can be efficiently stored, retrieved, exported, and inferred. In this paper, we give an example of a bibliographic platform based on BIBFRAME ontology to verify the proposed partition algorithm.

There is still a lot to be studied and improved upon in the future. For example, the case in this paper does not involve rule reasoning, which will be added in subsequent research. Reasoning is regarded as one of the core functions of semantic web. Another job that needs to be improved is how to set the *k* value. Currently, *k* is selected on the basis of experience, which depends too much on the human factor. Finally, we will use the method of machine learning to automatically detect the optimal *k* value.

ACKNOWLEDGEMENT

The research is granted financial support from National Social Science Fund of China (19BTQ024).

References

Erkimbaev, A. O., Zitserman, V. Y., Kobzev, G. A., Serebrjakov, V. A., & Teymurazov, K. B. (2013). Publishing scientific data as linked open data. *Scientific and Technical Information Processing*, *40*(4): 253-263. DOI:10.3103/S014768821304014X

Craig A. Knoblock, Pedro Szekely, Eleanor Fink, Duane Degler, David Newbury, Robert Sanderson, ... Yixiang Yao (2017). Lessons learned in building linked data for the American art collaborative. *in Proc. The Semantic Web - ISWC* 2017, 263-279. DOI:10.1007/978-3-319-68204-4_26

Chen Tao, Zhang Yongjuan, Liu Wei, & Zhu Qinghua (2019). Several specifications and recommendations for the publication of linked data. *Journal of Library Science in China*, *45*(1):34-46

Mohammad Farhan Husain, Pankil Doshi, Latifur Khan, & Bhavani Thuraisingham (2009). Storage and retrieval of large RDF graph using Hadoop and MapReduce. *CloudCom 2009,* LNCS 5931, 680-686. DOI:10.1007/978-3-642-10665-1_72

Kurt, R., & Richard, E. S. (2010). High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store. *in Proc. Programming Support Innovations for Emerging Distributed Applications*, ACM, 4:1-4:5. DOI:10.1145/1940747.1940751

Khushboo, T., & Abhishek B. (2017). A review of large-scale RDF document processing in Hadoop MapReduce framework. *International Journal of Scientific Research Engineering & Technology (IJSRET)*, *6*(2):123-126. .

Vaibhav Khadilkar, Murat Kantarcioglu, Bhavani Thuraisingham, & Paolo Castagna (2012). Jena-HBase: a distributed, scalable and efficient RDF triple store. *in Proc. International Semantic Web Conference (ISWC),* Springer, 1-4.

Nikolaos, P., Ioannis, K., Dimitrios, T., & Nectarios K. (2012). *H_2RDF*: adaptive query processing on RDF data in the cloud. *in Proc. 21st International Conference on World Wide Web*, 397-400. DOI:10.1145/2187980.2188058

Alfredo, C., Rajkumar, B., Vincenzo P., & Giovanni P. (2017). MapReduce-basedalgorithms for managing big RDF graphs: state-of-the-art analysis, paradigms, and future directions. *in Proc. 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 898-905. DOI:10.1109/CCGRID.2017.109

Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, & Zhongyuan Wang (2013). A distributed graph engine for web scale RDF data. *Proceedings of the VLDB Endowment, 6*(4):265-276. DOI:10.14778/2535570.2488333

Rong Gu, Wei Hu, & Yihua Huang (2014). Rainbow: a distributed and hierarchical RDF triple store with dynamic scalability. *in Proc. IEEE International Conference on Big Data*, 561-566. DOI:10.1109/BigData.2014.7004274

Yingjie Li, & Jeff Heflin (2010). Query optimization for ontology-based information integration. *Proceedings of the 19th ACM Conference on Information and Knowledge Management, CIKM 2010*, Toronto, Ontario, Canada. DOI:10.1145/1871437.1871623

Razen AI-Harbi, Yasser Ebrahim, & Panos Kalnis (2014). PHD-Store: an adaptive SPARQL engine with dynamic partitioning for distributed RDF repositories. *CoRR*.

Ruben ,V., Miel, V.S., & Pieter, C. (2014). Web-scale querying through Linked Data Fragments. *Proceedings of the 7th Workshop on Linked Data on the Web*

Huang, J. W., & Daniel J. A. (2016). LEOPARD: lightweight edge-oriented partitioning and replication for dynamic graphs. *Proceedings of the VLDB Endowment, 9*(7):540-551. DOI:10.14778/2904483.2904486

Wang, R. & Kenneth, C. (2012). A graph partitioning approach to distributed RDF stores. *in Proc. IEEE 10th International Symposium on Parallel and Distributed Processing with Application (ISPA)*, 411-418. DOI:10.1109/ISPA.2012.60

Yun Hao, Gaofeng Li, Pingpeng Yuan, & Hai Jin (2017). An association-oriented partitioning approach for streaming graph query. *Scientific Programming, 11*:1-11. DOI:10.1155/2017/2573592

About the authors

Dr. Tao Chen is a postdoctoral researcher in Nanjing University and Shanghai Library. His academic background in computer science and electrical engineering has influenced his research interests, focusing on data science, linked data, ontology and semantic web. He has more than 20 papers published.

Rongrong Shan is PhD candidate at the Department of Library, Information Science & Archive at Shanghai University.

Dr. Hui Li is a postdoctoral in Shanghai Library. She is a recent PhD graduate in Computer Science. Her research interests are in the area of social network analysis and natural language processing.

Dongsheng Wang is currently a PhD candidate in Computer Science at University of Copenhagen, he received master's degree from Korea University. His research interests are in the areas of big data, semantic web and data mining.

Wei Liu (aka. Keven Liu) is the Deputy Director of Shanghai Library and Institute of Scientific and Technological Information of Shanghai. He is an adjunct professor of Fudan University and Shanghai University in Shanghai, China. He took part in many major digital library projects in China since 1995. He is very active in developing digital humanities infrastructures in recent years.