Graph Data Transformations in Gradoop

Matthias Kricke[1], Eric Peukert[1], Erhard Rahm[1]

Abstract: The analysis of graph data using graph database and distributed graph processing systems has gained significant interest. However, relatively little effort has been devoted to preparing the graph data for analysis, in particular to transform and integrate data from different sources. To support such ETL processes for graph data we investigate transformation operations for property graphs managed by the distributed platform Gradoop. We also provide initial results of a runtime evaluation of the proposed graph data transformations.

Keywords: Graph analytics; Big Data; Graph transformations; Data integration

1 Introduction

The flexible and scalable analysis of large amounts of graph data has gained significant interest in the last decade and is supported by graph database systems (e.g., Neo4j), graph extensions in relational DBMS and a growing number of distributed platforms including those based on Apache Spark and Flink like GraphX, Gelly or Gradoop [Ju17a]. A largely neglected topic, however, is the support for ETL-like operations to prepare the graph data for analysis which requires to transform data sources into the supported graph format, to consolidate different graphs and to integrate them into a combined graph. As in traditional analysis platforms these steps can be highly complex and easily require the majority of time for graph analytics.

We have begun to investigate ETL and data integration for graph data for (extended) property graphs managed by the distributed open-source graph processing platform Gradoop [Ju16, Ju18]. Gradoop provides already different connectors to import data from relational databases or CSV files into property graphs. Furthermore, we provide initial match approaches within the FAMER system [SPR17, SPR18] to match and cluster graph vertices derived from multiple data sources. In this paper, we propose additional Gradoop operations to transform graphs to facilitate their integration with other graphs or to make them better suitable for analysis. For example, a bibliographic network with publications and their authors might have to be transformed for an easier analysis of co-authorships, e.g., by generating a graph with author vertices and co-authorship edges only. The proposed

[1] University of Leipzig, ScaDS Dresden Leipzig, Augustusplatz 10, 04109 Leipzig, Germany, {kricke, peukert, rahm}@informatik.uni-leipzig.de

doi:10.18420/btw2019-12

operations are not only relevant for Gradoop but should be useful for other platforms supporting property graphs.

We thus extend Gradoop with a number of generic transformation operations that can be used to define advanced graph transformations on property graphs. Each graph transformation can implicitly trigger a series of low-level graph changes (e.g. vertex/edge/property additions and deletions) that do not need to be defined by the user. Composite transformations can be expressed from basic ones and the transformations we propose are implemented with Apache Flink for parallel execution and good scalability to large graphs.

After a discussion of related work, we briefly introduce Gradoop and outline the overall data integration process. We then describe and illustrate the new operations for graph data transformation. In Section 5, we present initial results of a runtime evaluation before we conclude.

2 Related Work

Graph preparation and transformation have received little attention in research so far especially with reference to integration and analysis of property graphs. On the other hand, there have been some algebraic, declarative and imperative approaches for graph transformation some of which have been considered for graph data processing. Algebraic approaches have been used for model transformation in software engineering [Lö93, Eh97] and more recently for the parallel execution with the vertex-centric processing model of Pregel [KTG14, Ar10, Ma10] or Map Reduce [Be15].
Declarative approaches rely on a declarative language like Cypher and Sparql to query, construct and transform graphs. Cypher and Sparql provide only limited support for graph transformation with its RETURN statement (Cypher) and CONSTRUCT statement (Sparql) but new language proposals like Open Cypher [Gr18] and G-Core [An18] provide extensions to express graph grouping and aggregations. BigGra [TH17] translates an SQL-like query language called UnQL+ to the Pregel processing model in GraphX. In this approach navigational queries and transformations are expressed as so-called structural recursions which seem complex to be defined by a user. Some additional proposals provide a declarative specification of graph extraction from relational databases, for example expressed with a Datalog-based language as done in GraphGen [XKD15] or Table2Graph[Le15] that is based on MapReduce.
Imperative methods provide languages for a step-wise definition of graph transformations. A well known representative is Gremlin/TinkerPop [Ro15] offering a set of low-level operators for navigation and traversal of graphs and for adding or removing vertices and edges. The GraphBuilder tool [Ja13] mainly focuses on the construction of graphs rather than their transformation based on the extraction of values from data sources. The support for the actual transformation is limited to filters. GraphGen [XD17], GraphX and Gelly[KVH18] also provide only limited support for transformation, letting the user add edges, vertices and attributes manually for each necessary transformation.

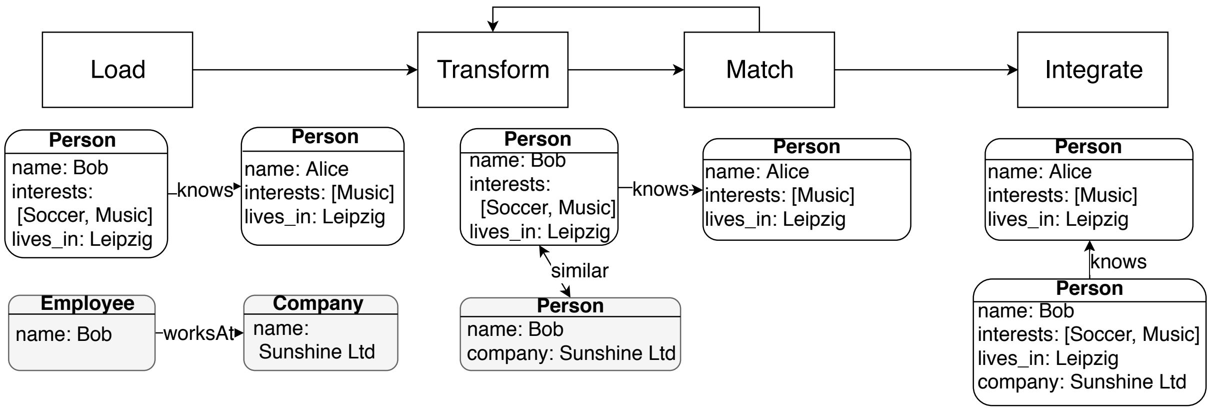

Fig. 1: Sample graph data integration pipeline combining person data from a social network and a company database.

3 Background

3.1 Graph Data Model

Gradoop [Ju16, Ju18] is based on an extended version of the *property graph model* (PGM) [RN10, RN12] which is widely used in graph database systems (e.g. Neo4j) and parallel processing systems such as Apache Spark GraphX. A property graph is a directed multigraph supporting properties and type labels for both vertices and edges. The properties are represented by key-value pairs (e.g. *name* : *Bob*). Properties are defined at the instance level and no schema definition is necessary. In extension to the PGM Gradoop supports storage and analysis of multiple property graphs called *logical graphs* that can also have a label and properties. Properties can be atomic (string, numeric, boolean, etc.) or collection-valued, e.g. lists. Gradoop also supports a number of generic operators on graphs and graph collections (for pattern matching, subgraph filtering, etc.) that can be used within workflows for graph analysis. The workflows can be specified in a declarative domain-specific language called GrALa. The implementation of the Gradoop operators is built on Apache Flink to achieve a parallel execution and scalability to large graphs.

3.2 Graph Data Integration

Figure 1 shows a typical data integration pipeline to integrate several sources into a graph for further analysis. Initially, already existing graphs are loaded or data from different sources such as databases or files of different formats (e.g. CSV, JSON, XML) are transformed into property graphs. The individual graphs may then have to be transformed to achieve a similar graph structuring and to facilitate further integration steps. In the example of Figure 1, we simplify the second graph by transforming the company vertices into properties of

Operator	**GrALa**
Property To Vertex	`graph.propertyToVertex(label, propertyName, newLabel, newPropertyName, edgeConfig, condense)`
Vertex to Property	`graph.propagateToNeighbor(label, edgeConfig)`
Vertex To Edge	`graph.vertexToEdge(vertexLabel, newEdgeLabel)`
Edge To Vertex	`graph.edgeToVertex(edgeLabel, newVertexLabel, edgeLabelSourceToNew, edgeLabelNewToTarget)`
Connect Neighbors	`graph.connectNeighbors(vertexLabel, edgeDirection, neighborVertexLabel, newEdgeLabel)`
Invert Edge	`graph.invertEdge(label, newLabel)`
Cluster Fusion	`graph.fuse(fusionConfig)`
Grouping	`graph.groupBy(vertexGroupingKeys, edgeGroupingKeys)`
Cypher Construct	`graph.query(patternQuery, constructionQuery)`

Tab. 1: Overview of structural graph transformation operators in Gradoop. Italic operations are new.

person vertices. To integrate the different graphs, we have to identify matching vertices and edges that need to be fused together. Given that the graphs may contain vertices and edges of many different types this is typically a complex process that is currently under investigation. What has already been implemented is the FAMER system [SPR17, SPR18] to link and cluster equivalent entities from multiple graphs, e.g., the vertices for the same person in the example. Clustered entities are fused together to create a single vertex in the integrated graph with the combined property values (e.g., for person *Bob* in the example). The integrated graph can be further transformed to support specific analytical purposes.

4 Graph Transformation

Table 1 gives an overview on the implemented transformations in Gradoop that change the graph structure. Additional simpler transformations include *property transformations* to change properties based on an UDF or basic functions like string splitting or concatenation. Due to space restrictions we only describe the first five transformations in more detail in the following. The grouping and Cypher operations have already been described in earlier work [JPR17, Ju17b]. Grouping allows us to determine structural graph aggregations with super-vertices and super-edges summarizing several vertices and edges based on common label and property conditions. Gradoop also has initial Cypher support to specify construction patterns such that the found instances for a query pattern can be transformed. We omit the description of the invert edge operation since it is simple and only inverts the edge direction plus the label of the edge. The cluster fusion operation combines several equivalent vertices into one and takes the union of different properties and combines different values for the same properties based on a specified function, e.g., to prefer the longest string or values from preferred sources (similar to fusion operations for relational data [BN09]).

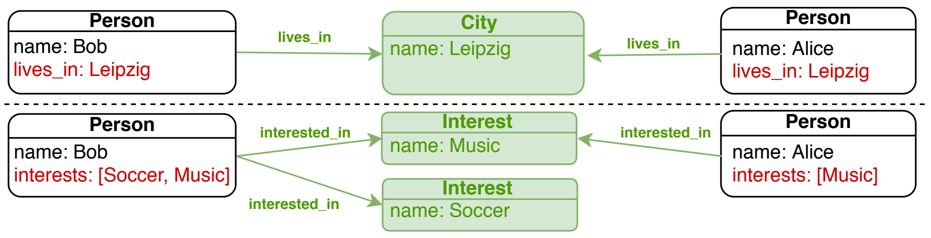

Fig. 2: PropertyToVertex examples for atomic (top) and collection (bottom) properties. Newly created/removed elements are shown in green/red, respectively.

In general, it is desirable that one can reverse the effect of graph transformations unless they lead to a reduced information such as deletes or grouping. We therefore have pairs of transformations and their inverse operations (`propertyToVertex` and `VertexToProperty`, `EdgeToVertex` and `VertexToEdge`). Furthermore we have to deal with both atomic and collection-based properties as well as with the creation / avoidance of duplicate information.

4.1 Property to Vertex & Vertex to Property

Property extraction is one of the most basic transformations and expressed in GrALa as: `graph.propertyToVertex(`$label$`,` $propertyName$`, newLabel, newPropertyName, edgeConfig, condense)`. It applies to all vertices of a given $label$ and for the values of property $propertyName$ it creates new vertices with label $newLabel$ and property $newPropertyName$. For collection-based values a vertex is created for every value in the collection. Note that this operator is very beneficial to transform data imported from data files (e.g., in CSV format) into a property graph. In this case, one can generate a vertex per input record and then generate additional vertices and connecting edges for selected properties by applying `propertyToVertex`.

Since the same property value can occur many times, the creation of new vertices can lead to many duplicate vertices. We can thus choose to avoid such duplicate vertices (parameter $condense$). Such vertices can thus be connected to multiple originating vertices and thus represent shared information. The deduplication with the $condense$ option is limited to equal values and thus only covers clean data sources. Hence, an additional deduplication for the created vertices may become necessary to fuse equivalent vertices with different names.

The user has several options to connect a newly created vertex to the original vertex with parameter $edgeConfig$: no edge, origin to new, new to origin and bidirectional. If an edge is created a user-defined label is set which is defined in the $edgeConfig$. The upper example in Fig. 2 shows the extraction of the atomar property $lives\_in$. Alice and Bob are living in the same city and therefore the $City$ vertex is deduplicated based on the value of the property $name$. All vertices Leipzig originates from are the start point of an edge with the label $lives\_in$ while the $City$ vertex is the target of these edges. The $edgeConfig$ for this

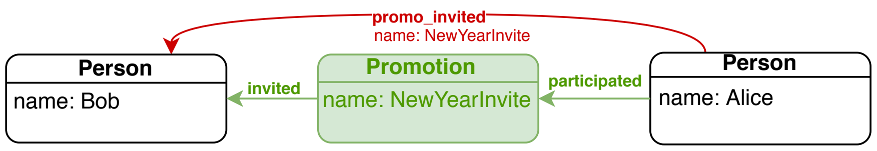

Fig. 3: The edge *promo_invited* is replaced (red) by the *Promotion* vertex with two edges (green) by using the *Edge to Vertex* operator. The *Vertex to Edge* operator inverts this operation.

example is: (label: *lives_in*; direction: *origin to new*). For the second example of Figure 2, we have a shared interest in *Music* so that this new vertex is connected to both *Bob* and *Alice*. This representation is obviously much better suited to analyze shared interests than the use of interest properties.

The inverse operation *vertex to property* is relatively straight-forward. Here *label* and *edgeConfig* are used to select the vertices to transform as well as the target vertices where the new properties should be added. As a result we can reverse both transformations shown in Figure 2.

4.2 Edge to Vertex & Vertex to Edge

The *Edge to Vertex* operator is beneficial for already existing graph structures. It creates a new vertex and two new edges for every edge with the desired label in the graph. GrALa code: `graph.edgeToVertex(edgeLabel, newVertexLabel, edgeLabelSourceToNew, edgeLabelNewToTarget)`. For the example in Figure 3 the GrALa call would be: `graph.edgeToVertex(`*promo_invited*`, `*Promotion*`, `*participated*`, `*invited*`)`. The operator turns the edge *promo_invited* between Alice and Bob with all its properties into a vertex and adds two user-defined edges.

The intuitive counterpart to *Edge to Vertex* is *Vertex to Edge*. It converts vertices with a specified label into edges between adjacent vertices. In GrALa it is defined as: `graph.vertexToEdge(vertexLabel, newEdgeLabel)`. To identify the necessary edges we select the "middle vertex" v with the specified label *vertexLabel* (which is to be replaced) and compose every direct neighbor with an edge going to v with every neighbor with an outgoing edge from v. With the *Vertex to Edge* operation we can revert the example given in Figure 3. The sole entry in the source set is Alice and in the target set Bob. Hence, an edge is created between Alice and Bob with the new label and all properties.

4.3 Connect Neighbors

The operator *Connect Neighbors* is designed to create relations between same-type vertices sharing a common neighbor vertex, e.g., employees of the same company (Fig.

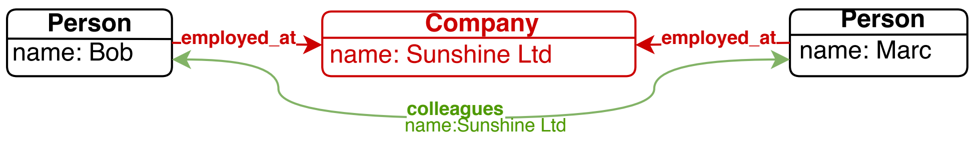

Fig. 4: The example shows two *Person*s that got connected by there shared *Company*.

4) or authors of the same publication. In GrALa, the operator call is expressed by using: `graph.connectNeighbors(vertexLabel, edgeDirection, neighborVertexLabel, newEdgeLabel)`. Here, *vertexLabel* is the label of the central (shared) vertex and *neighborVertexLabel* the label of the vertices to be connected. Parameter *edgeDirection* is interpreted with respect to the shared vertex and can either be incoming, outgoing or undirected. For each pair of indirectly connected vertices of type *neighborVertexLabel* a new bidirectional edge of type *newEdgeLabel* is created. This is in contrast to the *Vertex to Edge* operation that creates directed edges and can be applied to vertices between neighbors of different type as shown in the example of Fig. 3.

Figure 4 shows an example where Bob and Marc share the same *Company* vertex. Therefore, the vertex and the two edges pointing to it are removed in favor of the *colleagues* edge which retains the properties of the removed vertex. If several edges are created, each of those edges is containing the property set of the removed vertex. The operation can also be used to create a co-author network from a publication network as discussed in the introduction.

5 Evaluation & Discussion

For the evaluation of the operators we used Flink version 1.5.0, Gradoop version 0.4.1 and a subset of the OpenAcademicGraph (as of 2017-06-09, MAG files 1 - 59)[Si15, Ta08]. OpenAcademicGraph contains two bibliographic datasets: the MAG dataset with 166 million and the the AMiner dataset with 155 million publication records. Each line in the datasets contains the Json representation of one publication.

Our subset contains 60 million publication records from the MAG dataset and comprises 102,5 GB of unpacked data. We read the data with the JsonDataImport of Gradoop that resulted in a so-called *Initial* evaluation dataset of 60 million vertices. However, some of our operators (*Edge to Vertex*, *Connect Neighbors*) require the data to already contain edges and relations between vertices. We thus created a second dataset called *Extended* using the *PropertyToVertex* operation. We extracted the properties author, affiliation, keywords and field of study all with condense option activated. This resulted in 100 million vertices and 74 million edges. Note that this graph may still include some deduplication problems not covered by *PropertyToVertex*, e.g., due to different variations of author or affiliation names.

We executed our benchmark operations on a Shared Nothing cluster of the Leipzig University Computing Center. We used 9 nodes of the cluster where each had 2 sockets equipped with

	Atomic Property Extraction			Collection Property Extraction		
Configuration	min	average	max	min	average	max
no edge, no condense	821s	838.2s	857s	851s	866.1s	894s
no edge, with condense	829s	846.7s	875s	853s	873.8s	899s
create edge, no condense	848s	853.3s	860s	910s	1066.7s	1235s
create edge, with condense	924s	940.6s	955s	905s	927.1s	955s

Tab. 2: Runtimes of atomic and collection property extraction.

6 core CPUs (Intel Xeon E5-2620 v3, 2.4 GHz, supports Hyperthreading), 128GB RAM, 6 SATA hard disks with 4 terabyte each and 10 Gigabit/s Ethernet interface. One node was designated to be the master node while all others where configured to work with 96 task executors in total (12 each). Each tested operator was executed at least 10 times and measurements contain I/O.

Table 2 shows the runtimes of the *PropertyToVertex* operator for atomic and collection-based property extraction. We consider the impact of whether or not edges are created between newly created and original vertices as well as whether or not property values are deduplicated (*condense* options). For the atomic case, we consider the *venue* properties and for the collection-based extraction we use the *FoS* properties (Field of Study) of publications. We observe that there are mostly only small differences between atomic and collection-based extraction. In the atomic case 22 million *venue* vertices and edges are created without deduplication and only 10 million vertices with deduplication. The collection-based property extraction created 40 million *FoS* vertices and edges without and 12.5 million vertices with deduplication. The highest runtime is in the collection-based scenario where more than 50 million new graph elements are created. Deduplication with the *condense* options typically incurs only a small additional runtime. For the collection-based extraction it is even faster since the much lower number of vertices to be created more than outweighed the deduplication effort.

Operator	minimum	average	maximum
Vertex to Property	1449s	1528.5s	1635s
Vertex to Edge	1055s	1088s	1147s
Edge to Vertex	69s	72,3s	76s
Connect Neighbors	1964s	2148.6s	2296s
Invert Edges	63s	66.65s	70s

Tab. 3: Runtimes of structural transformations.

The evaluation of the remaining transformations are based on the second dataset and Table 3 shows the resulting runtimes. We observe that edge-based transformations achieve the lowest runtimes favored by a small number of properties for edges. This makes communication in the cluster and creating new objects less expensive. Furthermore, the transformations can be implemented using only the vertex Ids. In contrast, operators like *Vertex to Edge*, *Connect Neighbors* or *Vertex to Property* rely on the graph structure and the whole graph needs to be loaded. *ConnectNeighbors* turned out to be most expensive. For our evaluation this operator creates the co-author connections between authors of the same paper.

6 Conclusion & Future Work

We proposed structural transformation operations for property graphs with simple or collection properties to facilitate data integration and graph analysis. The operations have been implemented with Apache Flink and added to the open-source platform Gradoop. An initial evaluation for bibliographic data showed the applicability and relative efficiency of the operators. In future work we will further evaluate and optimize graph transformation operators and their use in real application scenarios as well as for graph-based data integration, in general.

7 Acknowledgements

This work was funded by the German Federal Ministry of Education and Research within project ScaDS Dresden/Leipzig (BMBF 01IS14014B). Computations for this work were done with resources of Leipzig University Computing Center.

References

[An18] Angles, R. et al.: G-CORE: A core for future graph query languages. In: Proc. ACM SIGMOD. pp. 1421–1432, 2018.

[Ar10] Arendt, T.; Biermann, E.; Jurack, S.; Krause, C.; Taentzer, G.: Henshin: advanced concepts and tools for in-place EMF model transformations. In: MODELS. pp. 121–135, 2010.

[Be15] Benelallam, A.; Gómez, A.; Tisi, M.; Cabot, J.: Distributed Model-to-model Transformation with ATL on MapReduce. In: Proc. ACM SIGPLAN. pp. 37–48, 2015.

[BN09] Bleiholder, Jens; Naumann, Felix: Data fusion. ACM Computing Surveys (CSUR), 41(1):1, 2009.

[Eh97] Ehrig, H. et al.: Algebraic approaches to graph transformation. In: Handbook Of Graph Grammars And Computing By Graph Transformation: Volume 1: Foundations, pp. 247–312. World Scientific, 1997.

[Gr18] Green, A.; Junghanns, M.; Kiessling, M.; Lindaaker, T.; Plantikow, S.; Selmer, P.: openCypher: New Directions in Property Graph Querying. In: Proc. EDBT. 2018.

[Ja13] Jain, N. et al.: Graphbuilder: scalable graph ETL framework. In: Proc. 1st GRADES Workshop. 2013.

[JPR17] Junghanns, Martin; Petermann, André; Rahm, Erhard: Distributed grouping of property graphs with GRADOOP. Proc. BTW conf., 2017.

[Ju16] Junghanns, M.; Petermann, A.; Teichmann, N.; Gómez, K.; Rahm, E.: Analyzing extended property graphs with Apache Flink. In: Proc. SIGMOD Workshop on Network Data Analytics. 2016.

[Ju17a] Junghanns, M. et al.: Management and analysis of big graph data: current systems and open challenges. In: Handbook of Big Data Technologies, pp. 457–505. Springer, 2017.

[Ju17b] Junghanns, Martin; Kießling, Max; Averbuch, Alex; Petermann, André; Rahm, Erhard: Cypher-based graph pattern matching in GRADOOP. In: Proc. GRADES workshop. 2017.

[Ju18] Junghanns, M.; Kießling, M.; Teichmann, N.; Gómez, K.; Petermann, A.; Rahm, E.: Declarative and distributed graph analytics with GRADOOP. PVLDB, 11(12), 2018.

[KTG14] Krause, C.; Tichy, M.; Giese, H.: Implementing graph transformations in the bulk synchronous parallel model. In: FASE. 2014.

[KVH18] K., Vasiliki; V., Vladimir; H., Seif: High-Level Programming Abstractions for Distributed Graph Processing. IEEE Trans. Knowl. Data Eng., 30(2):305–324, 2018.

[Le15] Lee, S.; Park, H.; Lim, S.; Shankar, M.: Table2Graph: A Scalable Graph Construction from Relational Tables Using Map-Reduce. In: Proc. IEEE Conf. Big Data Computing Service and Applications. pp. 294–301, 2015.

[Lö93] Löwe, M.: Algebraic approach to single-pushout graph transformation. Theoretical Computer Science, 109(1-2):181–224, 1993.

[Ma10] Malewicz, G.; Austern, M.; Bik, A.; Dehnert, J.; Horn, I.; Leiser, N.; Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proc. ACM SIGMOD. 2010.

[RN10] Rodriguez, M.; Neubauer, P.: Constructions from dots and lines. Bulletin of the American Society for Information Science and Technology, 36(6):35–41, 2010.

[RN12] Rodriguez, M.; Neubauer, P.: The graph traversal pattern. In: Graph Data Management: Techniques and Applications, pp. 29–46. IGI Global, 2012.

[Ro15] Rodriguez, M. A.: The Gremlin Graph Traversal Machine and Language. In: Proc. Symp. Database Programming Languages. pp. 1–10, 2015.

[Si15] Sinha, A.; Shen, Z.; Song, Y.; Ma, H.; Eide, D.; Hsu, BJ.; Wang, K.: An overview of Microsoft Academic Service (MAS) and applications. In: Proc. WWW. 2015.

[SPR17] Saeedi, A.; Peukert, E.; Rahm, E.: Comparative evaluation of distributed clustering schemes for multi-source entity resolution. In: Proc. ADBIS conf. pp. 278–293, 2017.

[SPR18] Saeedi, A.; Peukert, E.; Rahm, E.: Using Link Features for Entity Clustering in Knowledge Graphs. In: Proc. ESWC. 2018.

[Ta08] Tang, J.; Zhang, J.; Yao, L.; Li, J.; Zhang, L.; Su, Z.: Arnetminer: extraction and mining of academic social networks. In: Proc. ACM SIGKDD. pp. 990–998, 2008.

[TH17] Tung, L.; Hu, Z.: Towards systematic parallelization of graph transformations over Pregel. Int. Journal of Parallel Programming, 45(2):320–339, 2017.

[XD17] Xirogiannopoulos, K.; Deshpande, A.: Extracting and Analyzing Hidden Graphs from Relational Databases. Proc. ACM SIGMOD, pp. 897–912, 2017.

[XKD15] Xirogiannopoulos, K.; Khurana, U.; Deshpande, A.: GraphGen: Exploring Interesting Graphs in Relational Data. PVLDB, 8(12):2032–2035, 2015.