Towards a property graph generator for benchmarking

Arnau Prat-Pérez
Joan Guisado-Gámez
Xavier Fernández Salas
{aprat|joan|xavierf}@ac.upc.edu
DAMA-UPC, Universitat Politècnica de Catalunya

Petr Koupy
Siegfried Depner
Davide Basilio Bartolini
{petr.koupy|siegfried.depner|davide.bartolini}@oracle.com
Oracle Labs

ABSTRACT

The use of synthetic graph generators is a common practice among graph-oriented benchmark designers, as it allows obtaining graphs with the required scale and characteristics. However, finding a graph generator that accurately fits the needs of a given benchmark is very difficult, thus practitioners end up creating ad-hoc ones. Such a task is usually time-consuming, and often leads to reinventing the wheel. In this paper, we introduce the conceptual design of DataSynth, a framework for property graphs generation with customizable schemas and characteristics. The goal of DataSynth is to assist benchmark designers in generating graphs efficiently and at scale, saving from implementing their own generators. Additionally, DataSynth introduces novel features barely explored so far, such as modeling the correlation between properties and the structure of the graph. This is achieved by a novel property-to-node matching algorithm for which we present preliminary promising results.

ACM Reference format:
Arnau Prat-Pérez, Joan Guisado-Gámez, Xavier Fernández Salas, Petr Koupy, Siegfried Depner, and Davide Basilio Bartolini. 2017. Towards a property graph generator for benchmarking. In *Proceedings of Graph Data-management Experiences & Systems, Chicago, May 19th 2017 (GRADES'17),* 6 pages.
DOI: http://dx.doi.org/10.1145/3078447.3078453

1 INTRODUCTION

During the last decade, the amount of available data has grown exponentially and it is expected to grow even more over the next years. Much of these data present themselves in the form of property graphs, which are graphs whose vertices and edges are labeled and have associated properties in the form of key-value pairs. The increasing popularity of property graphs has provoked the irruption of many systems specialized on their management [1, 3] and analysis [2, 9, 22], as well as benchmarking initiatives to fairly compare them [5, 11–13, 18].

One of the difficulties of evaluating graph systems is to obtain representative datasets with the desired scale and characteristics – because data is often sensitive and business critical, and companies do not disclose them. Thus, the use of synthetically generated graphs has become a common practice among graph-oriented benchmark designers.

Recent literature on graph system benchmarking reveals an increasing interest on large synthetic graphs that can reliably mimic real datasets at both the structural and property value levels. On the one hand, it is acknowledged that the structure of a graph can heavily affect the performance and behavior of an algorithm [20, 23]. On the other hand, graph-based technology is penetrating into domains such as social networks, mobility planning or drug development, just to cite a few. Each domain requires application specific benchmarks with graphs where not only the structure is relevant (i.e. it must be similar to that of the graphs of the domain), but also the distributions of the property values and the way these properties are correlated with the underlying graph structure. In many real graphs, we observe property-structure correlations in the form of joint probability distributions between the property values of pairs of connected nodes [17]. The presence of these property-structure correlations can be determinant for the performance of some queries. This is an aspect accurately modeled, for instance, by the modern LDBC Social Network Benchmark [11], which uses correlated graphs that have been crafted after a detailed choke point analysis similar to that done on more traditional benchmarking such as TPC-H [7].

Given these trends, we foresee an increasing need for synthetic graph generators that can produce large graphs that are realistic both structurally and in terms of properties. However, most of existing graph generators only focus on the structure [8, 10, 14], and those that generate properties are designed for specific use cases [11, 24]. Implementing a property graph generator is a time consuming task, thus we need tools to save practitioners from such a burden.

In this paper we present the conceptual design of DataSynth, a work-in-progress domain agnostic graph generation framework, for the generation of property graphs for benchmarking at a scale. DataSynth assumes a shared-nothing environment and borrows techniques from existing tools to generate data efficiently in parallel. At the core of DataSynth lies a novel and fundamental property-to-node matching algorithm that allows decoupling the generation of properties from the generation of the graph structure, while preserving property-structure correlations. According to our first experiments, this approach looks promising. Summarizing, DataSynth is designed to be capable of:

- Generating property graphs using configurable schemas that consist of multiple node and edge types and properties.

GRADES'17, Chicago

978-1-4503-5038-9/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3078447.3078453

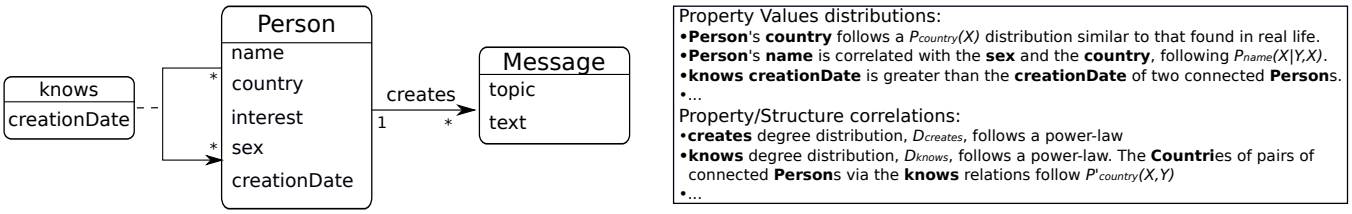

Figure 1: Running Example

- Reproducing user-provided property value distributions and property-structure correlations, similar to those observed in many real graphs.
- Scaling to billion edge graphs and be work-efficient by applying in-place data generation and other optimization techniques whenever possible.

2 GRAPH GENERATOR REQUIREMENTS

In this section, we identify and classify data requirements related to the **schema**, the **structure** and the property **distributions**; and functional requirements related to the **scale factor** of the graph and **other** characteristics of a property graph generator. We use the running example of Figure 1, which represents a simple social network.

Schema. Graph-based algorithms are being adopted in many domains, from recommender systems to social network analysis, route planning, etc. All these applications rely on property graphs that exhibit a myriad of different schemas that graph generators have to be able to reproduce. Following the typical property graph model, such schemas are usually defined in terms of the *node* and *edge* types, their associated *properties* and the *cardinality* of the edge types. Thus, a property graph generator should allow expressing a schema in similar terms.

For instance, in our running example there are two node types, `Person` and `Message`, and two edge types, `knows` and `creates`. `Person` has five properties: `name`, `country`, `interest`, `sex` and `creationDate`. `Message` has two: `topic` and `text` and the edge `creates` has one: `creationDate`. Without loss of generality, we will assume that all properties in this schema are of type String. Regarding the cardinality of the edges, `knows` is a $^*\rightarrow^*$ relationship between `Persons` and `creates`, which is a $1\rightarrow^*$ relationship between `Persons` and the `Messages`.

Structural. Graph theory defines tens of structural properties to characterize graphs, such as number of connected components, clustering coefficient, degree distribution, centrality, diameter, assortativity, community distribution, etc. Graphs from different domains exhibit differences in such structural characteristics, which can affect the performance of the algorithms. Thus a property graph generator should be able to generate graphs reproducing them.

For instance, our running example imposes a structural characteristic, namely $\mathcal{D}_{knows}$, over the `knows` edge that should be able to reproduce. In addition, the pairs of `countries` of connected `Persons` by `knows` should follow a joint probability distribution $P_{country}(X,Y)$ reflecting that `Persons` from the same `Country` are more likely to know each other. Consequently, the resulting graph will be divided into communities of `Persons` from the same `Country`.

Distribution. The distribution of property values in real graphs is rarely uniform. For example, our running example, `Person`'s `country` should follow a distribution similar to that found in real life. Moreover, property values may be correlated with each other. For instance, the `name` of a `Person` is clearly correlated with the `sex` and the `country`. Finally, other relations may exist between the values of different properties, such as binary logical relations between numerical values. For instance, in our running example, the `knows creationDate` should be greater than the `creationDate` of two connected `Person`'s by means of the edge.

Scale Factor. Existing benchmarks usually define some sort of scale factors for their data. Each scale factor is used to size the capabilities of the systems with respect to the amount of processed data. Some existing benchmarks base such scale on the number of nodes of the graph [18], others prefer the number of edges [12] or a combination of nodes and edges [13], or even on the size on disk of the datasets [11]. Thus, a property graph generator should provide different means of specifying the scale of the produced graph.

Other requirements. We have identified a series of other characteristics that we believe any graph generator should have. Specially relevant is its scalability and efficiency, which have to allow the generator to produce large graphs, as those found in real-life. Also, taming a cross-domain property graph generator with such a degree of flexibility requires a properly designed interface. This should include some sort of Domain Specific Language (DSL) for the specification of the data to generate, with the corresponding syntax completion tools. Finally, beside the interface, generators should provide connectors for integrating the framework with production-level technologies such as databases and cluster storages (e.g. HDFS).

3 RELATED WORK

Table 1 summarizes the state of the art generators in relation to the requirements described in Section 2. Note that in the table, marked cells indicates that the generator allows configuring explicitly the corresponding aspect.

The LDBC Social Network Benchmark [11] (LDBC-SNB) models a realistic social network, with multiple node types (Persons, Posts, Topics, etc.) and edge types (knows, creates, has). Among the novel features it incorporates, is specially remarkable the generation of a friendship graph with property-structure correlations, which also has several desirable properties observed in social networks such as a realistic community structure [19], a small diameter, a large clustering coefficient and a Facebook-like degree distribution. However it does not provide many ways to change the produced schema, but some distributions and cardinalities can be tuned using configuration files.

Myriad [4] is a domain-agnostic property graph generator for structured relational data. It is flexible and allows the definition

	Schema					Structure	Distributions		Scale Factor			Others		
	Node		Edge		Edge		Property values distribution	Property Structure correlation	Node	Edge	Node + edge	Scalability	Language	Integrability
	type	prop.	type	prop.	cardinality									
LDBC-SNB [11]					x	dd, cc	x		x	x	x	x		x
Myriad [4]	x	x	x		x[1]	dd			x			x	xml	
gMark [6]	x		x		x	dd			x	x	x		xml	
RMat [8]						pl dd			x			x		
LFR [16]						pl dd, c			x					
BTER [14]						dd, accd			x			x		
Darwini [10]						dd. ccdd			x			x		

Table 1: Related work summary. In Structure, dd: degree distribution, cc:cluster coefficient, pl: power law, c: communities, accd: average clustering coefficient per degree, ccdd: clustering coefficient distribution per degree. x[1]: supports 1→1 & 1→*.

of different domain objects with multiple properties, including foreign keys, which can be seen as one-to-one or one-to-many edges. Myriad does not allow generating many-to-many relationships, thus cannot be applied to fully model property graphs. Additionally, Myriad implements in-place data generation using pseudo-random number generators, a technique we borrow for DataSynth.

gMark [6] is a recent property graph generator that allows specifying diverse schemas and configure some of the graph characteristics. However, the schema definitions are restricted to just different nodes and edge types and properties are omitted. Moreover, the configurable structural characteristics are limited to degree distributions and does not allow specifying complex graph structures.

RMat is a graph generator used in the Graph-500 competition [18] and produces graphs with a power-law degree distributions. Similarly, the LFR graph generator not only generates power-law degree distributions but also communities of nodes. This graph generator is typically used to benchmark community detection algorithms, since the communities are known beforehand.

The BTER graph generator goes beyond degree distributions and is also capable of reproducing the average clustering coefficient per degree of an input graph. As a side effect of its generation process, BTER produces graphs with a positive degree of assortativity and a community structure. Darwini [10] extends BTER and captures the clustering coefficient distribution at a finer granularity. Both BTER and Darwini are highly scalable, which allows the generation of Facebook-scale graphs in the order of a trillion of edges. Additionally, BTER and Darwini produce graphs with a small diameter due to its generation process, although this cannot be configured.

4 DATASYNTH

In this Section, we describe how DataSynth approaches the problem of property graph generation, given the requirements identified in Section 2. Figure 2 summarizes how conceptually DataSynth generates property graphs. First the schema is received in a domain specific language (DSL), that allows expressing all the needs identified by the **schema**, **structural**, **distributions** and **scale factor** requirements. Then, for each edge type, we generate node properties and graph structure independently, which are later matched (node ids are assigned to graph structure nodes) in order to reproduce the required joint probability distributions specified by the user. Finally, the properties of the edges are generated.

We follow this approach for the following reasons. Building a graph generator capable of configuring all the existing structural characteristics yet generating properties at the same time, that also reproduces the joint probability distributions between property values of nodes, is a very complex task. Moreover, we do not even know which of the structural characteristics have an actual impact on the performance of the algorithms, which may actually depend on the domain of the generated graph and the type of queries to perform. For instance, while it is acknowledged that the degree distribution and diameter affect the performance of some algorithms such as BFS, impact of the community structure or the degree of assortativity is not yet assessed. Thus, our approach lets the user to choose between existing structure generators and structural properties they reproduce, fulfilling the **structural** requirement and keep the framework open to advances in the field.

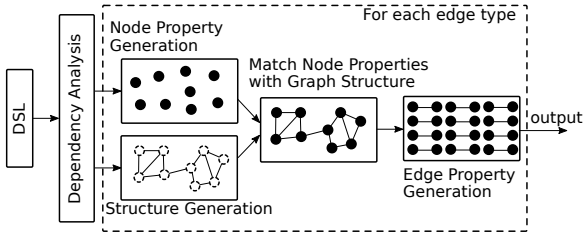

Figure 2: DataSynth general approach

In the rest of the section, we detail the whole approach. We first introduce some preliminary concepts, namely the data model, property generators and structure generators, and continue with the actual property graph generation process. We do not detail the design of the DSL because it is not in the scope of this paper.

4.1 Preliminaries

Data Model. DataSynth is designed with scalability in mind, for that purpose we rely on distributed tables as the data storage. In more detail, we use one "Property Table" (PT), which is a 2-column table [id:Long, value:type], for each pair <node type, property> and <edge type, property>. In our running example, it would create eight PTs. In addition, we use one "Edge Table" (ET), which is a 3-column table [id:Long, tailId:Long, headId:Long], for each edge type. The first column is used to identify an edge instance, while the second and third columns contain the *ids* of the nodes connected by the edge. The *ids*, either of nodes or edges, are unique per type, and range between 0 and $n-1$, where n is the number of instances of the given type. For our running example, DataSynth would create three ETs.

Property Generators (PGs). PGs are pluggable "objects" that can be referenced from the DSL to specify the way property values are generated. A PG implements an interface with the following methods:

- $initialize : (\dots) \rightarrow void$

Sets up the state of the PG. It takes a variable number of parameters that depend on the strategy to generate the data (e.g. a filename to load a dictionary).
- $run : (id : Long, r(id) : Long, \dots) \rightarrow T$
  It generates the property values of the instances. It takes two parameters, i) the *id* of the instance, either of a node or an edge, for which it generates the property value and ii) the result of calling $r(id)$ which is a deterministic function that generates a random number using *id*. Optionally, it also takes a variable number of parameters used to specify the correlation between property values.

Notice that *run* depends exclusively on the *id* and the result of a deterministic function called with the same *id*. This allows regenerating a property value in-place by just knowing the *id*, for instance, in different computing nodes. This approach is the same to that used in Myriad, where the function $r()$ is a pseudo-random number generator (PRNG) with skip seed. Such a PRNG implements efficiently a method $r : (i : Long) \rightarrow Long$ that returns the ith random number in a sequence. The PG can use the number returned by $r()$ to generate a property value randomly. In order to ensure independence between properties, DataSynth builds a different $r()$ for each PT. Additionally, passing the *id* to *run* allows the generation of user-controlled *uuids* that can be correlated with other properties such as the time.

Note that the interface of a PG is flexible enough to allow the generation of the properties of our running example. The optional number of parameters of the *run* method allows implementing the generation of sequences that follow probability or conditional probability distributions. For example, the *run* method of PG_{name}, the PG for `Person name`, which depends on `Person country` and `Person sex`, has the signature $run : (id, r(id), String, String) \rightarrow String$. In order to generate `names` with a realistic given distribution, that method can implement the "Inverse Transform Sampling" using the provided random number. This allows fulfilling part of the **distribution** requirement.

Structure Generators (SGs). Similar to PGs, SGs can be provided by users to customize the generation of the graph structure (the edges), and are referenced from the DSL as well. SGs implement the following interface.

- $initialize : (\dots) \rightarrow void$
  Initializes the SG, similarly to PG.It takes a variable number of parameters that depend on the strategy to build the structure (e.g. a file with an empirical degree distribution).
- $run : (n : Long) \rightarrow ET$
  Generates an ET with the edges of a graph of size *n* (number of nodes). The values of $tailId$ and $headId$ range between zero and $n - 1$, while the ids of the edges range between zero and $m - 1$, where *m* is the ET size, which depends on the generation process.
- $getNumNodes : (numEdges : Long) \rightarrow Long$
  Returns the number *n* of nodes to call the method *run(n)* such that the resulting ET is of size *numEdges*.

This approach allows accommodating state-of-the-art graph generators such as BTER or Darwini. Their parameters –in this case the degree and clustering coefficient distributions– would be passed to the method *initialize*. A call to the *run* method would then generate the structure for the given number of nodes. Finally, the method *getNumNodes* would be used to specify the scale of the graph in terms of the number of edges.

4.2 Property Graph Generation Process

The data generation process begins analyzing the schema described by the user to reveal dependencies among the data to be generated. In more detail, from the dependencies analysis we get a dependency graph, which we traverse to preserve the dependencies between the tasks. This guarantees that the required parameters are available for each task when we execute it. There are three different types of tasks: generate property, generate graph and match graph.

The dependency analysis is required for the following reason. Generate property and generate graph tasks take the size of the task to run as an input (e.g. the number of node instances or the edges of the graph to generate). These sizes are sometimes given by the size of the output of another task. For example, imagine that the user of our running example only defines the scale factor of the graph by means of the number of `Persons` to be N, but says nothing about the rest of entities. Thus, how many instances of `Message` does DataSynth have to generate? Notice that each `Message` depends on the number of instances of the edge `creates` (due to the $1 \rightarrow *$ cardinality). In turn, the number of edges `creates` follows $\mathcal{D}_{creates}$, a degree distribution observed in real-life and provided by the user, and is conditioned by N. Thus, to infer the number of `Messages`, we need first to generate the structure for the edge `creates`. Once the structure of `creates` is generated, its size determines the number of `Messages` to create. Finally, we can apply the match operator between `Persons`, `Messages` and `creates`. Notice that the chain of dependencies can be much complex than the one used in this example.

Alternatively, the user could be interested in specifying the scale of the graph in terms of the number of edges `creates`, instead of the number of `Persons` nor the number of `Messages`. In this case, DataSynth would use the *getNumNodes* method with the desired number of edges as a parameter, and use the result to size the graph structure and the number of `Persons`. The size of the resulting graph structure would be used to determine the number of `Messages`. This flexibility in specifying the scale of the graph lets DataSynth to fulfill the **scale** requirement.

Generate Structure. This task is responsible of generating the graph structure of a given edge type. For each edge type, the user specifies the SG to use and its parameters. DataSynth initializes the SG and calls the *run* method, which returns a table with the graph structure. The number of nodes used to call the run method is determined either by the user or from the dependency analysis of DataSynth. Remember that this task generates a graph whose ids must be matched later with node ids to reproduce the desired correlations (if any).

Generate Property. Properties, either from nodes or edges, are created by calling the *run* method of its corresponding PG *pg*, which is initialized with the parameters specified by the user using the method *initialize*. The *run* method is called *n* times, which is the size of the PT *p* willing to generate. Before the generation of *p*, its corresponding PRNG *r* is initialized as well. For those properties that are not correlated with any other property, the ith row of *p* is $[i, pg.run(i, r(i))]$. For those properties correlated with other properties is $[i, pg.run(i, r(i), val_0, \dots, val_k)]$, where val_j is the result of

calling the *run* method for the generation of the jth property the currently generated property depends on (using the appropriate PG and PRNG). Such call, in turn, can lead to successive calls of other run methods. The dependency analysis guarantees that the recursion terminates. For example, in order to generate the property `sex` of `Person`, we would call:

$$pg_{sex}.run(i, r_{sex}(i), pg_{country}.run(i, r_{country}(i)))$$

Properties for edges can be similarly generated, using the *ids* of the endpoints of the edge if needed. Note that thanks to our approach where property values are generated independently, these can be generated efficiently in parallel in a distributed system by just knowing the *ids* of the nodes to generate. Knowing such *ids* is easy, because we just need the number of instances which are unique per type and not globally. This allows achieving the desired scalability expressed in the **others** requirement.

Graph Matching. The task of graph matching consists in matching entries of a PT p with the nodes of the generated graph structure g, in such a way that the desired property-structure correlation is preserved. We model the property-structure correlation as a joint probability distribution $P(X, Y)$. This distribution, which is provided by the user, expresses the probability of picking a random edge of the graph and observing property values X and Y in its endpoints. In those cases where an edge type is not correlated with any property, the matching is done randomly.

The input of the graph matching is: the PT p of the property that is correlated with the structure, a joint probability distribution $P(X, Y)$ and a graph structure g . The goal is to find a mapping function f that maps the node ids of the graph structure to ids of the PT, such that the observed $P'(X, Y)$ after applying the mapping function f is as close as possible as $P(X, Y)$. This is the way we fulfill the remaining of the **distribution** requirement.

We approach the problem using the *Stochastic Block Model (SBM)*. SBM is a model used for graphs where there are groups of entities with a given property value or category (one group per property value). In SBM, for each pair of groups $< i, j >$, there is a probability $\delta_{i,j}$ that an edge exists between each pair of members of the two groups (i and j can be the same). The SBM is typically used to study community detection algorithms.

For example, suppose that in our input PT p, there are n different property values. Let $Q = \{q_0, \ldots, q_{n-1}\}$ be the frequencies of each of the values observed in p (which are the sizes of each group). Let W be a $n \times n$ matrix such that W_{ij} is the number of edges between the nodes of group i and j[1]. Given $P(X, Y)$ and the number of edges m of the graph structure, we can compute $W_{ii} = \frac{2mP(i,i)}{q_i(q_i-1)}$ and $W_{ij} = \frac{2mP(i,j)}{q_i q_j}$ if $i! = j$. In other words, our problem is equivalent to classifying the nodes of g into n groups of sizes $Q = \{q_0, \ldots, q_{n-1}\}$, in such a way that the intra and inter-group edges are as close as possible to those in W. Then, function f is built by assigning to each node of g an *id* out of those of p that have the value corresponding to the partition the node has been assigned. Thus, the problem can be seen as a graph partitioning problem.

To solve this graph partitioning problem, we have implemented a variation of the LDG streaming graph partitioning algorithm [21]. In LDG, a node arrives along with its edges, and is placed to that partition where lay most of its neighbors already seen (weighted by a factor depending on the remaining capacity of the partition). In our case, instead of taking the decision based on the node's degree, we place the node to the partition t that minimizes the Frobenius Norm between W_T and W:

$$\arg\min_t \parallel W_t - W \parallel_F^2, \tag{1}$$

where W_t is the $n \times n$ matrix where $W_{t,ij}$ contains the number of edges connecting nodes with properties i and j, given that we put the current node to partition t. As in LDG, the score is balanced by the remaining capacity $(1 - \frac{s_t}{q_t})$, where s_t is the number of nodes placed to partition t so far. We name this method as SBM-Part. Notice that a small variation of SBM-Part can also be applied to bi-partite graphs, since the SBM can model this type of graphs as well. If the bi-partite graph is between two different node types, the input would contain two PTs instead of one.

Preliminary evaluation of graph matching. We conducted some preliminary experiments to assess the quality of the proposed graph matching. We generated a set of graphs using the LFR and RMAT graph generators. We have configured LFR with an average degree of 20, a maximum degree of 50, a minimum community size of 10 and a maximum community size of 50, which are the parameters used in [15]. The mixing factor is set to 0.1. The rest of parameters have been left to their default values. We have generated graphs of sizes 10k, 100k and 1M nodes. In the case of the RMAT, we have used the default parameters. We have generated graphs of scale 18, 20 and 22.

We partitioned each of the graphs g into k groups representing k different values, using LDG. The size of the ith group is $n \cdot \frac{max(geo(0.4,i),1/k)}{\sum_{j=1}^{k} max(geo(0.4,j),1/k)}$, where n is the number of nodes of g and *geo* is a geometric distribution with parameter 0.4 in this case. We use a geometric distribution to emulate real-life graphs, where groups have different sizes. Then, the nodes of the ith partition were assigned the property value i. Then, we computed our joint probability distribution $P(X, Y)$ empirically. Finally, we created a PT p with *ids* between 0 and $n - 1$, containing as many rows with property value i as the size of partition i. Then, we run SBM-Part using p, $P(X, Y)$ and g. We sent the nodes to SBM-Part randomly.

Figure 3 and 4 show the CDF of the expected ($P(X, Y)$) and observed ($P'(X, Y)$) distributions after running SBM-Part for different graphs and number of k values. The x axis corresponds to the different pairs of values $< i, j >$, and are sorted by decreasing probability in the expected CDF, for both distributions.

In Figure 3, we fix the number of k values to 16, and vary the size of the graph for the two generators. We see two interesting results. The first is that the quality of the results for LFR graphs seems to be very good, with the observed distribution with a shape that is very similar to the expected, and better than that obtained for RMAT graphs. For the latter, however, note that SBM-Part is able to reproduce the pronounced slope at the beginning of the distribution, which in general correspond to those entries of $P(X, Y)$ where $X = Y$. The results suggest that the performance of the algorithm might be affected by the structure of the graph being partitioned. The second result is that the quality of the results does not seem to be affected by the size of the graphs, which suggests that the method could scale to larger graphs qualitatively speaking.

[1]We work with absolute number of edges instead of probabilities for convenience

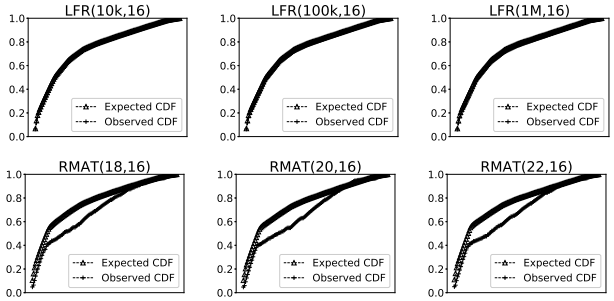

Figure 3: Results for LFR and RMAT graphs of different sizes and 16 different values

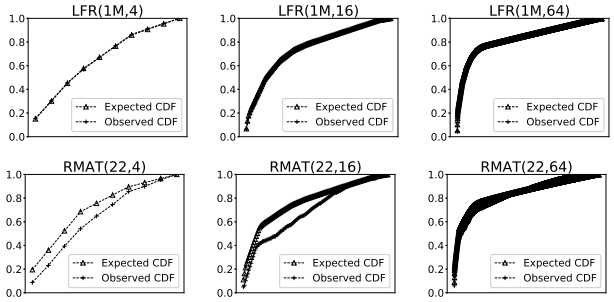

Figure 4: Results for LFR and RMAT graphs of fixed size and different number of values

In Figure 4, we see the results when we fix the sizes and change the number of k to 4, 16 and 64. The results are very similar to those observed in the previous experiments. The method works consistently very well with LFR graphs while for RMAT graphs, it seems that the larger the number of values the better. This seems to confirm that there is a strong influence of the structure of the graph to the quality of the results. Finally, about the performance of the algorithm, it takes about 1100s to process the largest problem, RMAT-22 (with 67M of edges) and 64 values, using a single thread on an Intel Xeon E-2630 v3 at 2.4Ghz. No optimizations of any kind have been implemented.

5 DISCUSSION AND FUTURE WORK

We have presented the design of DataSynth, a framework for property graph generation with user-defined property distributions and correlations, different graph structures and property-structure correlations. The method relies on a novel graph partitioning algorithm, called SBM-Part, that allows matching properties to nodes in a graph, in such a way that desired joint probability distributions are preserved.

SBM-Part is a greedy algorithm that does not guarantee an optimal solution, thus strict constraints cannot be fully guaranteed. However, special cases of one-to-one and one-to-many edges could be efficiently handled by more specific and efficient operators. These, would generate both the property values and the graph structure at the same time, which would boost performance and allow reproducing strict constraints reliably. Other specific graph structures such as trees, which appear in message cascades in social networks, might require also special strategies. In this case, information propagates through the cascade, which could be modeled using a vertex-centric approach that propagates the information through the cascade iteratively.

We have presented preliminary results of SBM-Part. Further study is required, including a complexity analysis, optimization strategies, etc. Specially interesting would be understanding which is the relation between the graph structure and the provided joint probability distribution (i.e. in which situations the algorithm performs well and which does not). Additionally, performing experiments for multi-valued properties would also be interesting.

Finally, more work is needed regarding scalable graph generators with realistic structural characteristics. So far, BTER and Darwini are the structural graph generators that allow tweaking a larger spectrum of structural characteristics. Studying which of the characteristics are important for a given domain, and building scalable graph generators to reproduce these characteristics are still open problems.

ACKNOWLEDGMENTS

DAMA-UPC thanks the Ministry of Economy, Industry and Competitiveness of Spain, Generalitat de Catalunya, for grant numbers TIN2013-47008-R and SGR-1187 respectively and also the EU H2020 for funding the Uniserver project (ICT-04-2015-688540). Also, thanks to Oracle Labs for the support to our research on graph technologies.

REFERENCES

[1] Neo4J Graph Database. www.neo4j.com. (????).
[2] Oracle Parallel Graph Analytics. http://www.oracle.com/technetwork/oracle-labs/parallel-graph-analytics. (????).
[3] Sparksee Graph Database. www.sparsity-technologies.com. (????).
[4] Alexander Alexandrov, Kostas Tzoumas, and Volker Markl. 2012. Myriad: scalable and expressive data generation. *PVLDB* 5, 12 (2012), 1890–1893.
[5] Timothy G Armstrong et al. 2013. LinkBench: a database benchmark based on the Facebook social graph. *SIGMOD* (2013), 1185–1196.
[6] Guillaume Bagan et al. 2017. gMark: schema-driven generation of graphs and queries. *TKDE* 29, 4 (2017), 856–869.
[7] Peter Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. *TPCTC* (2013), 61–76.
[8] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. *SDM* (2004), 442–446.
[9] Avery Ching et al. 2015. One trillion edges: Graph processing at facebook-scale. *PVLDB* 8, 12 (2015), 1804–1815.
[10] Sergey Edunov et al. 2016. Darwini: Generating realistic large-scale social graphs. *arXiv:1610.00664* (2016).
[11] Orri Erling et al. 2015. The LDBC social network benchmark: Interactive workload. *SIGMOD* (2015), 619–630.
[12] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web* 3, 2 (2005), 158–182.
[13] Alexandru Iosup et al. 2016. Ldbc graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *PVLDB* (2016), 1317–1328.
[14] Tamara G Kolda et al. 2014. A scalable generative graph model with community structure. *SISC* 36, 5 (2014), C424–C452.
[15] Andrea Lancichinetti and Santo Fortunato. 2009. Community detection algorithms: a comparative analysis. *Physical review E* 80, 5 (2009), 056117.
[16] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. 2008. Benchmark graphs for testing community detection algorithms. *Physical review E* 78, 4 (2008), 046110.
[17] Miller McPherson, Lynn Smith-Lovin, and James M Cook. 2001. Birds of a feather: Homophily in social networks. *Annual review of sociology* 27, 1 (2001), 415–444.
[18] Richard C Murphy et al. 2010. Introducing the graph 500. *Cray Users Group (CUG)* (2010).
[19] Arnau Prat-Pérez and David Dominguez-Sal. 2014. How community-like is the structure of synthetically generated graphs? *GRADES* (2014), 1–9.
[20] Arnau Prat-Pérez et al. 2011. Social based layouts for the increase of locality in graph operations. *DASFAA* (2011), 558–569.
[21] Isabelle Stanton and Gabriel Kliot. 2012. Streaming graph partitioning for large distributed graphs. *SIGKDD* (2012), 1222–1230.
[22] Narayanan Sundaram et al. 2015. GraphMat: High performance graph analytics made productive. *PVLDB* 8, 11 (2015), 1214–1225.
[23] Hao Wei et al. 2016. Speedup Graph Processing by Graph Ordering. *SIGMOD* (2016), 1813–1828.
[24] Chengcheng Yu et al. 2014. On efficiently generating realistic social media timeline structures. *SSDBM* (2014), 45.