EGG: A Framework for Generating Evolving RDF Graphs

Karim Alami, Radu Ciucanu, and Engelbert Mephu Nguifo

Université Clermont Auvergne & CNRS LIMOS, France
alami.karim7@gmail.com, ciucanu@isima.fr, mephu@isima.fr

Abstract. We demonstrate EGG (**E**volving **G**raph **G**enerator), an open-source framework for generating evolving RDF graphs based on finely-tuned temporal constraints given by the user. During the demonstration, we will showcase the highly-expressive constraints that the user can specify in EGG to generate evolving graphs over various real-world use cases, the accuracy and scalability of the generator, and the ease of using EGG in performance comparisons of evolving graph processing systems.

1 Introduction

Large-scale RDF graphs are used to model a variety of real-world domains. In practice, both nodes and edges of such graphs have properties that are naturally evolving over time. For example, in a geographical database storing information about cities and transportation facilities, the nodes of type city have evolving properties e.g., weather and air quality, whereas the edges that encode transportation facilities between cities have evolving properties e.g., price.

To be able to realize rigorous empirical evaluations of research ideas, the graph processing community needs tunable evolving graph generators, which are particularly useful whenever real-world graphs are unavailable for public use. The community has well-known synthetic RDF graph generators (e.g., [1,2,3]), very few evolving RDF generators (e.g., EvoGen [4] that extends LUBM [3]), but to the best our knowledge, there is no schema-driven evolving graph generator.

We demonstrate EGG (**E**volving **G**raph **G**enerator), an open-source[1] framework for generating evolving graphs based on finely-tuned temporal constraints given by the user. We depict the architecture of EGG in Fig. 1. We built EGG on top of gMark [2], a state-of-the-art static graph generator. EGG takes as input (i) an initial graph generated by gMark, and (ii) an evolving graph configuration that encodes how the evolving properties of the node types and edge predicates from a gMark configuration should evolve over time. The output of EGG is an RDF graph annotated with temporal information (in the spirit of [6]) that encodes a sequence of graph snapshots satisfying the constraints given by the user.

In Section 2, we present an overview of EGG, whereas in Section 3 we describe our demonstration scenarios. Due to the lack of space, we omit several details that can be found on the GitHub page[1] of EGG, together with the different use cases and data that we will use throughout our demonstration scenarios.

[1] https://github.com/karimalami7/EGG

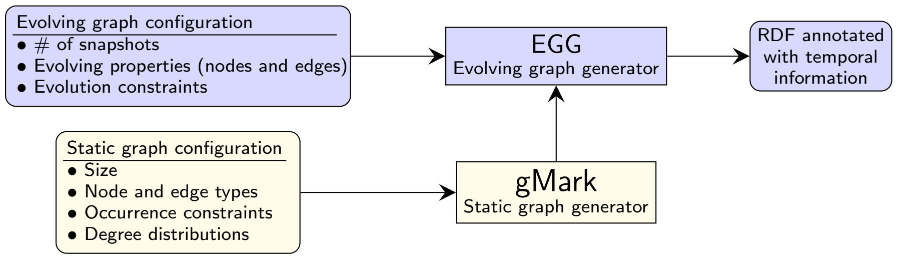

Fig. 1. Architecture of EGG. The bottom components are part of the existing gMark [2] static graph generator. The top components are part of our EGG contribution.

2 System Overview

In this section, we present gMark static graph configurations, EGG evolving graph configurations, and we briefly discus EGG implementation challenges. Similarly to gMark, EGG is schema-driven and domain-independent. We use next as running example a geographical database, but we have been additionally able to easily encode different domains such as a social network, a DBLP-like bibliographical network, or an online shop. All these schemas will be part of our demonstration.

Static graph configurations. Assume that a user wants to generate graphs simulating a geographical database storing data about cities, and different facilities such as transportation and hotels. The user can specify as gMark input the following types of constraints: (i) graph size, given as # of nodes; (ii) node types e.g., `city` and `hotel`, and edge types e.g., `train` and `contains`; (iii) occurrence constraints e.g., 10% of the graph nodes should be of type `city`, whereas 90% of the graph nodes should be of type `hotel`; (iv) degree distributions e.g.,

source type $\xrightarrow{\text{predicate}}$ *target type*	*In-distribution*	*Out-distribution*
`city` $\xrightarrow{\texttt{contains}}$ `hotel`	Uniform [1,1]	Zipfian

meaning that we can have an edge of type `contains` from a node of type `city` to a node of type `hotel`, with a Zipfian out-distribution (since it is realistic to assume that the number of hotels in a city follows such a power-law distribution) and a uniform [1,1] in-distribution (since a hotel is located in precisely one city).

We call such gMark graph configurations as being static since the nodes of type e.g., `city` and `hotel` are rarely created or deleted. Nonetheless, such nodes (as well as the different edges connecting them) possess properties that naturally evolve over time, in an interdependent manner. The user can specify such evolving properties as input of EGG, as we illustrate next.

Evolving graph configurations. Assume that our user generates with gMark a graph having nodes of type `city` and `hotel`, and edges of type `train` (connecting two cities) and `contains` (connecting a city to a hotel). Next, the user wants to add properties that evolve over time for the aforementioned nodes and edges, assuming that a graph snapshot corresponds to a day. We next give examples of such properties, together with finely-tuned constraints to evolve among consecutive snapshots. A node of type `hotel` has the following evolving properties:

– `availableRooms` (quantitative discrete), which can have as values integers in the interval [1,100], following a binomial distribution. There is a probability of 80% that it changes from a snapshot to the next one, and it can increment or decrement by an integer up to 5 between two consecutive snapshots.

– `star` (ordered qualitative), which can have five possible values, following a geometric distribution. It can only change every thirty snapshots, with a probability of 10%, and it can only increment or decrement by 1.

– `hotelPrice` (quantitative continuous), whose values follow a normal distribution in an interval that is dynamically constructed based on the value of the property `star`. Moreover, `hotelPrice` is anti-correlated with `availableRooms` i.e., if `availableRooms` decreases, then `hotelPrice` increases, and vice-versa.

The user can similarly specify evolving properties and evolution constraints for the node type `city` (e.g., properties `weather` and `airQuality`) and for the edge type `train` (e.g., property `trainPrice`). It is worth noting that we allow the EGG user to specify *validity properties* i.e., Boolean properties encoding whether a given node or edge exists at a given snapshot e.g., a train connection between two cities may not be valid during all snapshots.

Implementation challenges. Building a system like EGG is an ambitious goal since we allow the user to specify very expressive constraints. This leads to some interesting challenges that we briefly discuss next:

Computational complexity. As illustrated earlier, we allow the user to specify evolution constraints where the value of a property among consecutive snapshots depends on another property. We model the inter-dependencies between such evolving properties with a dependency graph. It is easy to see that if the aforementioned dependency graph is cyclic, the generation algorithm may not halt. Consequently, in our implementation we require that the dependency graph is acyclic and we sort it topologically to decide in which order we should apply the evolution constraints. Even for acyclic dependency graphs, we suspect that it is NP-complete to decide whether there exists a sequence of graph snapshots satisfying the input constraints. The exact complexity is an open question.

Storage redundancy. A naive solution to store the generated evolving graphs would be to entirely store each snapshot, which would yield a redundant storage due to the graph parts that are static throughout the snapshots. To minimize such redundancy, we rely on a storage format inspired by [6] that uses named graphs to express temporal information in RDF. Our output format (that we serialize using the TriG syntax[2]) allows us to decouple the storage of the static parts of the graph (i.e., structural information satisfied in all snapshots) and the evolving parts of the graph (i.e., the property values that change from a snapshot to the next one). For example, we use named graphs of the form

`ns1:G31 {<hotel:27> ns2:hasProperty <Property:availableRooms>.}`

encoding that a node of type `hotel` has a property `availableRooms`. Moreover, for each graph snapshot, we have a further named graph where each of the named graphs of the form above has associated a value e.g., `ns1:G31 ns3:value "57"`. We provide examples of such TriG output on the GitHub page of EGG.

[2] `https://www.w3.org/TR/trig/`

3 Demonstration Scenarios

During the demonstration, we will (i) introduce via examples the finely-tuned temporal constraints that the user can specify in EGG, (ii) emphasize the accuracy and scalability of EGG, and (iii) point out the ease of using EGG in performance comparisons of evolving graph processing systems.

(i) Finely-tuned constraints by example. We will show to the attendees how finely-tuned temporal constraints as those exemplified in Section 2 can be easily encoded in JSON as EGG input. In addition to our running example, we will also rely on several EGG real-world use cases: a social network in the spirit of the datasets used in [5], a DBLP-like co-authorship graph, an online shop in the spirit of WatDiv [1], and a university database in the spirit of LUBM [3]. All these use cases are also available online on the GitHub page of EGG.

(ii) Accuracy and scalability. For showing the accuracy of EGG and its sensitivity to different constraints, we will rely on the EGG visualization module to illustrate that the generated graphs match the input constraints. For example, we observe in Fig. 2 that the evolving properties satisfy the constraints in Section 2, in particular the anti-correlation between `hotelPrice` and `availableRooms`. As for the scalability of EGG, the attendees will generate graphs of increasing size or with an increasing number of snapshots, and observe that EGG has a linear time behavior. Detailed accuracy and scalability plots are available in the wikis of our GitHub page of EGG.

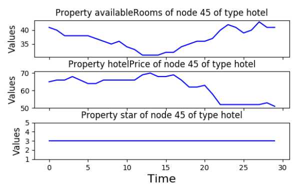

Fig. 2. Plots generated with EGG visualization module.

(iii) Impact on empirical evaluations. To emphasize the ease of realizing empirical evaluations on top of EGG, we will present a performance comparison of approaches for answering historical reachability queries [5], which ask whether there exists a path between two nodes in a specified interval of time. The attendees will generate evolving graphs with EGG and visualize the trade-offs between an algorithm found in [5] against a SPARQL implementation of our own on top of Apache Jena. We provide in a wiki on our GitHub page of EGG more details e.g., the data and queries to be used, and the types of generated plots.

References

1. G. Aluç and et al. Diversified stress testing of RDF data management systems. In *ISWC*, pages 197–212, 2014.
2. G. Bagan and et al. gMark: Schema-driven generation of graphs and queries. *IEEE TKDE*, 29(4):856–869, 2017.
3. Y. Guo and et al. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.
4. M. Meimaris and G. Papastefanatos. The EvoGen benchmark suite for evolving RDF data. In *MEPDaW/LDQ@ESWC*, pages 20–35, 2016.
5. K. Semertzidis and et al. TimeReach: Historical reachability queries on evolving graphs. In *EDBT*, pages 121–132, 2015.
6. J. Tappolet and A. Bernstein. Applied temporal RDF: efficient temporal querying of RDF data with SPARQL. In *ESWC*, pages 308–322, 2009.