Generating Flexible Workloads for Graph Databases

Guillaume Bagan
CNRS LIRIS
guillaume.bagan@liris.cnrs.fr

Angela Bonifati
University of Lyon 1 & CNRS LIRIS
angela.bonifati@univ-lyon1.fr

Radu Ciucanu*
University of Oxford
radu.ciucanu@cs.ox.ac.uk

George H. L. Fletcher
TU Eindhoven
g.h.l.fletcher@tue.nl

Aurélien Lemay
University of Lille 3 & INRIA
aurelien.lemay@inria.fr

Nicky Advokaat
TU Eindhoven
n.advokaat@student.tue.nl

ABSTRACT

Graph data management tools are nowadays evolving at a great pace. Key drivers of progress in the design and study of data intensive systems are solutions for synthetic generation of data and workloads, for use in empirical studies. Current graph generators, however, provide limited or no support for workload generation or are limited to fixed use-cases. Towards addressing these limitations, we demonstrate gMark, the first domain- and query language-independent framework for synthetic graph and query workload generation. Its novel features are: (i) fine-grained control of graph instance and query workload generation via expressive user-defined schemas; (ii) the support of expressive graph query languages, including recursion among other features; and, (iii) selectivity estimation of the generated queries. During the demonstration, we will showcase the highly tunable generation of graphs and queries through various user-defined schemas and targeted selectivities, and the variety of supported practical graph query languages. We will also show a performance comparison of four state-of-the-art graph database engines, which helps us understand their current strengths and desirable future extensions.

1. INTRODUCTION

Graph data management tools are rapidly evolving, as new features and capabilities are explored in response to practical demand and increasingly sophisticated user requirements. To better understand and drive forward this evolution, solutions for the generation of synthetic graph instances and query workloads are fundamental for the design of empirical studies of graph systems. Indeed, graph and query generation solutions are important for understanding the current limitations and potential improvements to graph management systems. Often in practice, graph database scenarios are driven by sophisticated query workloads. Fundamental examples include multi-query optimization, mapping discovery and query rewriting in data integration systems, and workload-driven graph database physical design [2].

Recently, there has been a flurry of works on graph database benchmarking [1, 4, 5], some of them addressing synthetic workload generation [1]. In our work, we extend this line of research by pushing forward the diversity and flexibility of graph and query workload generation.

In particular, we have designed and implemented gMark, a domain- and query language-independent framework targeting highly tunable generation of both graph instances and graph query workloads based on user-defined schemas. One of the most notable features of gMark is support for controlled schema-driven generation of fully diverse queries not only in terms of syntactic features but also in terms of expressiveness (in particular, recursive queries) and query selectivity (size of the query result as a function of the size of any input graph, reflecting the runtime behavior of queries). These features, which are not supported by current tools, permit fine-grained targeted testing of current graph data management systems, to better understand their performance, limitations, and optimization opportunities.

Contributions. We demonstrate gMark for graph and query generation. In summary, our system is highly tunable, featuring: (i) schema-driven data and query generation, (ii) multiple practical graph query languages (SPARQL, openCypher, SQL, Datalog) including recursion among the features, and (iii) selectivity estimation of generated queries. We will showcase graph instance and query workload generation through a set of highly tunable parameters and user-defined schemas. We will show that our system is able to simulate realistic graph domains and to fruitfully leverage user interactions. We will also present our generated instances and query loads at work with four current (commercial and open-source) graph database management systems.

The goal of the demonstration is to highlight the novel gMark capabilities and features. We will call attention to the importance of the continued study and development of domain- and query language-independent synthetic generation solutions, which are crucial in understanding the strengths and desirable directions for future research of graph data management systems. We present a system overview in Section 2 and our demonstration scenarios in Section 3. In our technical report [3] we focus more deeply on the gMark application to system performance comparison.

*Supported by EPSRC platform grant DBOnto.

Proceedings of the VLDB Endowment, Vol. 9, No. 13
Copyright 2016 VLDB Endowment 2150-8097/16/09.

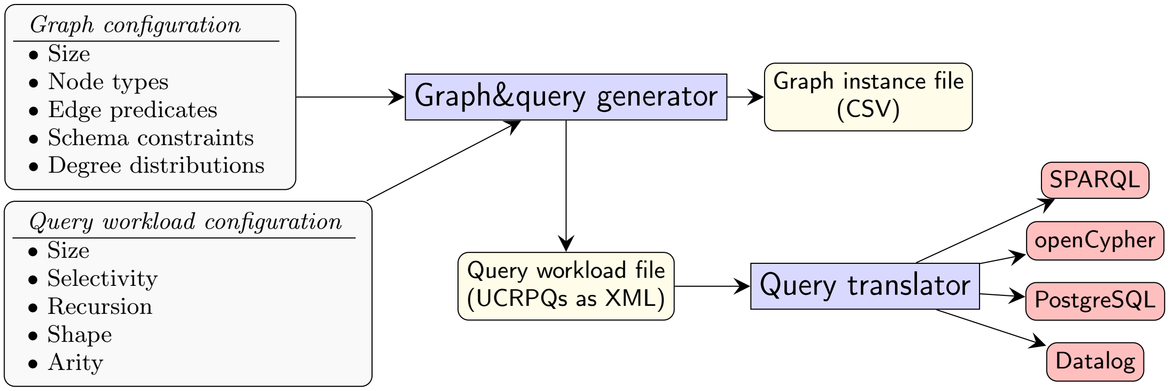


Figure 1: Overview of the gMark workflow.

2. SYSTEM OVERVIEW

Given a *graph configuration* and a *query workload configuration*, our system generates directed edge-labeled graphs and query workloads coupled to these graphs. Moreover, our query generator supports four concrete syntaxes, and can be easily extended to support additional practical query language syntaxes. The goal of this section is to highlight the main features and workflow of the system, as illustrated in Figure 1. We provide gMark as *open-source software*[1] for use in the graph processing community. Our system is implemented in C++ using the Standard Library.

2.1 Graph generation

Graph generation is driven by a *graph configuration*, which specifies constraints that the generated graphs should satisfy. The first parameter is the graph size, defined as the number of nodes. The next parameters (cf. Figure 1) encode a *graph schema* which allows the user to specify the sets of node types and edge labels; schema constraints (as proportions of a node type's or an edge label's occurrences or a fixed constant value); and, the in- and out-*degree distributions* of edge labels, with support for the following distributions: *uniform*, *Gaussian* (also known as normal), and *Zipfian*. For each distribution, the user can specify the relevant parameters (i.e., min and max for uniform, μ and σ for Gaussian, and s for Zipfian). If the user wants to specify only the in- or the out-distribution of an edge label, she can mark the other one as *nonspecified*.

Example 2.1 *Assume that we want to generate graphs simulating a bibliographical database that uses a simple schema consisting of 5 node types and 4 edge predicates. Intuitively, the database consists of researchers who author papers that are published in conferences (held in cities) and that can be extended to journals. We also want to specify constraints on the number of occurrences for both the node types and edge predicates, either as proportions of the total size of the graph or as fixed numbers (cf. Figure 2(a) and 2(b)). For instance, for graphs of arbitrary size, half of the nodes should be researchers, but a fixed number of nodes should be cities where conferences are held (in a realistic scenario the number of authors increases over time, whereas the number of cities remains constant). Then, we want to specify real-world relationships between types and predicates as in Figure 2(c). For instance, the first line encodes that the number of authors on papers follows a Gaussian distribution (the in-distribution of the schema constraint), whereas the number of papers authored by a researcher follows a Zipfian (power-law) distribution (the out-distribution of the schema constraint). The following lines in Figure 2(c) encode constraints such as: a paper is published in exactly one conference, a paper can be extended or not to a journal, a conference is held in exactly one city, the number of conferences per city follows a Zipfian distribution, etc. We can specify all aforementioned constraints in* gMark *via a few lines of XML.*

Node type	*Constr.*
researcher	50%
paper	30%
journal	10%
conference	10%
city	100 (fixed)

(a) Node types.

Edge predicate	*Constr.*
authors	50%
publishedIn	30%
heldIn	10%
extendedTo	10%

(b) Edge predicates.

source type $\xrightarrow{\textit{predicate}}$ *target type*	*In-distr.*	*Out-distr.*
researcher $\xrightarrow{\texttt{authors}}$ paper	Gaussian	Zipfian
paper $\xrightarrow{\texttt{publishedIn}}$ conference	Gaussian	Unif. [1,1]
paper $\xrightarrow{\texttt{extendedTo}}$ journal	Gaussian	Unif. [0,1]
conference $\xrightarrow{\texttt{heldIn}}$ city	Zipfian	Unif. [1,1]

(c) In- and out-degree distributions.

Figure 2: The bibliographical motivating example.

2.2 Query workload generation

Before introducing the parameters of the *query workload configuration* (cf. Figure 1), it is important to first present the abstract query language supported by gMark.

Query language. We focus on unions of conjunctions of regular path queries (UCRPQ), a fundamental language which covers many queries that appear in practice [7], e.g., the core constructs of SPARQL 1.1[2] (SPARQL with recursion) and openCypher[3]. Apart from recursive and non recursive graph queries, our language is expressive enough to cover typical analytic workloads including listing triangles, vertex neighborhood, and clique detection.

Given an input graph schema, let $\Sigma^+ = \{a, a^- \mid a \in \Sigma\}$, where Σ is the set of edge labels and a^- denotes the *inverse*

[1] https://github.com/graphMark/gmark

[2] http://www.w3.org/TR/sparql11-overview/

[3] http://www.opencypher.org/

of edge label a. Let $V = \{?x, ?y, \ldots\}$ be a set of variables and $n > 0$. A *query rule* is an expression of the form

$$(\overline{?v}) \leftarrow (?x_1, r_1, ?y_1), \ldots, (?x_n, r_n, ?y_n)$$

where: for each $1 \leqslant i \leqslant n$, it is the case that $?x_i, ?y_i \in V$; $\overline{?v}$ is a vector of zero or more of these variables, the length of which is called the *arity* of the rule; and, for each $1 \leqslant i \leqslant n$, it is the case that r_i is a regular expression over Σ^+ using $\{\cdot, +, *\}$ (i.e., *concatenation*, *disjunction*, and *Kleene star*). Without loss of generality, we restrict regular expressions to only use recursion (i.e., the Kleene star symbol $*$) at the outermost level. Hence, expressions can always be written to take either the form $(P_1 + \cdots + P_k)$ or the form $(P_1 + \cdots + P_k)^*$, for some $k > 0$, where each P_i is a *path expression* i.e., a concatenation of zero or more symbols in Σ^+. We refer to each rule pattern $(?x_i, r_i, ?y_i)$ as a *conjunct*. A UCRPQ *query* is a finite non-empty set of query rules, each of the same arity, with standard semantics following Datalog queries [6, 7]. In summary, a query is a collection of rules, each rule having several conjuncts, each conjunct having several disjuncts, and each disjunct being a path expression of a certain length.

Example 2.2 *The following query, where* `a`*,* `p`*, and* `e` *denote* `authors`*,* `publishedIn`*, and* `extendedTo`*, resp., selects all pairs of researchers and conferences/journals where researchers in their coauthorship network authored papers:*

$$(?x, ?z) \leftarrow (?x, (\mathtt{a}\cdot\mathtt{a}^-)^*, ?y), (?y, (\mathtt{a}\cdot\mathtt{p} + \mathtt{a}\cdot\mathtt{e}), ?z)$$

This query consists of one rule having two conjuncts. The second conjunct has two disjuncts, each of length two.

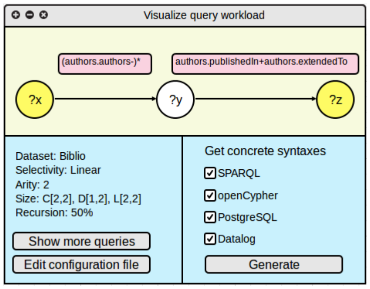
In our demo, our attendees will visualize this query (and other generated queries) as in Figure 3, and then its translation to the four supported query syntaxes as in Figure 4. Therefore, in the following we focus our attention on the translation of queries into the four syntaxes supported by gMark, with a particular focus on recursive queries.

In its first release, gMark supports the following practical query language syntaxes:

- openCypher, a popular graph query language;
- SPARQL 1.1, the W3C standard query language for linked graph data on the web;
- SQL:1999 recursive views, where we use the standard translation of UCRPQ's into recursive views, implemented using linear recursion [6, 7]; and
- Datalog [6].

We face two major challenges when translating queries in gMark's abstract syntax of UCRPQ's into queries in these practical languages. We next briefly discuss these.

The first challenge we face is dealing with differences in expressive power between the languages. UCRPQ's can be translated into equivalent queries in SPARQL, SQL, and Datalog, as we discuss below. The expressive power of openCypher, however, limits the class of queries which can be tested on systems implementing this language. First, openCypher does not support regular expressions having disjuncts under Kleene star where the path length is greater than one or there are occurrences of inverse symbols. Second, openCypher adopts an evaluation semantics which differs from the other three languages. In particular, openCypher enumerates only isomorphic embeddings of query rules in the database (i.e., variables in queries are mapped injectively into the data graph) whereas the other languages enumerate homomorphic embeddings (i.e., different variables may be mapped to the same node in the data graph). This difference in embedding semantics often leads to query output which differs dramatically from that returned by queries in the other languages. Since these two limitations (i.e., restrictions on syntax and semantics) are inherent features of openCypher, we cannot overcome them in our translation.



Dataset: Biblio
Selectivity: Linear
Arity: 2
Size: C[2,2], D[1,2], L[2,2]
Recursion: 50%

Get concrete syntaxes
☑SPARQL
☑openCypher
☑PostgreSQL
☑Datalog

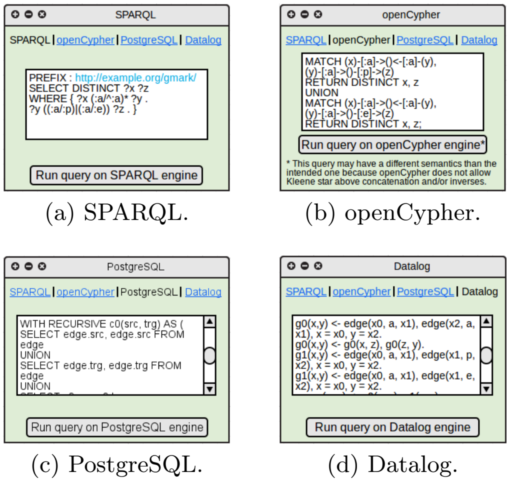
Figure 3: Visualization of the query from Example 2.2. The projected variables are in yellow.

The second challenge we face is the differing levels of support for UCRPQ's in the syntax of the languages. SPARQL directly supports regular expressions as so-called "property paths", and hence the translation from UCRPQ's to SPARQL is lightweight. SQL, Datalog, and openCypher however, do not directly support regular expressions in their syntaxes. We illustrate the challenges of expressing UCRPQ's in these languages with SQL; similar issues arise in translations for Datalog and openCypher. In our translation, each conjunct gets translated as a temporary SQL view expressed using a `WITH RECURSIVE` clause, where the graph is stored in a table `edge(src, trg, label)`. For recursive conjuncts having a Kleene star we use right-linear recursion, that is, we recur on the tail of path [6, Chapter 15].

3. DEMONSTRATION SCENARIOS

Our demonstration scenario consists of four parts. First, we would like to present to the attendees the diversity of parameters for graph and query workload generation supported by gMark. Second, we will demonstrate the use of the query translator to generate queries in practical query languages, and we will compare their capabilities for expressing graph queries in a succinct and accurate manner. Third, the attendees will be able to effectively evaluate four state-of-the-art graph systems using generated graphs and queries. Fourth, we will share with the attendees some of the insights that we gained while empirically evaluating the four systems, such as their current strengths and their limitations (such as, for instance, their limited support of recursive queries). The goal of the last two scenarios is to demonstrate to the attendees that thanks to gMark's novel features, we can study in depth and highlight new findings about the performance of graph database engines.

1. Controlling graph and query diversity. In the first scenario, we will illustrate the highly-configurable parameters that the user can tune for the generation of graphs (e.g., size, schema constraints, degree distributions) and query workloads (e.g., number of conjuncts or disjuncts, length of paths, arity, recursion, selectivity estimation). All these parameters are specified via an XML configuration file. We


SPARQL
SPARQL | openCypher | PostgreSQL | Datalog
Run query on SPARQL engine

```
PREFIX : http://example.org/gmark/
SELECT DISTINCT ?x ?z
WHERE { ?x (:a/^:a)* ?y .
?y ((:a/:p)|(:a/:e)) ?z . }
```

(a) SPARQL.

openCypher
SPARQL | openCypher | PostgreSQL | Datalog
Run query on openCypher engine*

```
MATCH (x)-[:a]->()<-[:a]-(y),
(y)-[:a]->()-[:p]->(z)
RETURN DISTINCT x, z
UNION
MATCH (x)-[:a]->()<-[:a]-(y),
(y)-[:a]->()-[:e]->(z)
RETURN DISTINCT x, z;
```

* This query may have a different semantics than the intended one because openCypher does not allow Kleene star above concatenation and/or inverses.

(b) openCypher.

PostgreSQL
SPARQL | openCypher | PostgreSQL | Datalog
Run query on PostgreSQL engine

```
WITH RECURSIVE c0(src, trg) AS (
SELECT edge.src, edge.src FROM
edge
UNION
SELECT edge.trg, edge.trg FROM
edge
UNION
```

(c) PostgreSQL.

Datalog
SPARQL | openCypher | PostgreSQL | Datalog
Run query on Datalog engine

```
g0(x,y) <- edge(x0, a, x1), edge(x2, a,
x1), x = x0, y = x2.
g0(x,y) <- g0(x, z), g0(z, y).
g1(x,y) <- edge(x0, a, x1), edge(x1, p,
x2), x = x0, y = x2.
g1(x,y) <- edge(x0, a, x1), edge(x1, e,
x2), x = x0, y = x2.
```

(d) Datalog.

Figure 4: Visualization of the query from Figure 3 in four concrete languages. Notice that openCypher does not allow Kleene star above concatenation and/or inverse edges.

will rely on the bibliographical schema (cf. Section 2), the UniProt schema [3], and also on other realistic schemas that we encoded (e.g., modeling a social network or an online shop). The attendees will notice that the generation time depends on the density of the graphs yielded by these schemas. Moreover, the attendees will *visualize* the generated queries as in Figure 3 and check the accuracy of the *selectivity estimation* (recall that this is one of the main novel features) by (i) generating a query of a required selectivity, (ii) running the query with any of the compatible systems, (iii) comparing the actual selectivity with the one specified in the input configuration file.

2. Comparing the succinctness of the query languages. Next, the attendees will have the chance to visualize the generated queries in all supported query languages (SQL, SPARQL, Datalog, openCypher) as exemplified in Figure 4, and to import them in all supported systems. We will highlight the importance of our query translator to generate queries in a diversity of concrete languages, which is necessary to evaluate state-of-the art systems that do not share a common query language. The attendees will notice that some of these languages are able to express graph queries in a *succinct* manner since they have been designed specifically to query graphs (SPARQL and openCypher), whereas the general-purpose languages (PostgreSQL and Datalog) are more verbose. Interestingly, as the next scenarios show, the most efficient in practice is the Datalog engine, despite being a general-purpose system. Additionally, we will highlight that not all languages can express the generated queries in an *accurate* manner i.e., as discussed in Section 2.2, openCypher limits the use of recursion.

3. Evaluating graph database management systems. The goal of this third scenario is to make the attendees aware of how easy it is to design a *large-scale empirical evaluation* on top of our generated graphs and query workloads, while spanning over a high variety of systems and query languages. The attendees will generate queries according to scenarios aiming at varying specific query parameters, visualize and import them in the considered systems and generate performance plots. For instance, we present in Figure 5 a plot summarizing the results of four systems, by reporting the size of the result and the execution time. For this example, the attendees will notice that there is a strong correlation between the result size and the execution time, that PostgreSQL is slower than the others by up to two orders of magnitude, the graph system cannot be fairly compared because of its limited expressiveness (cf. Section 2.2), the SPARQL system is the fastest for small result sizes, and the Datalog system has a steady behavior and scales the best.

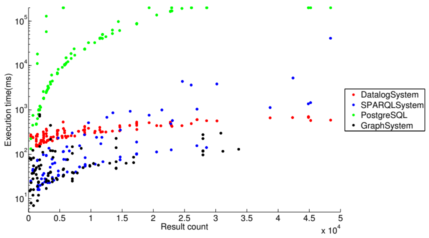


Figure 5: Scatter plot of result count versus execution time. The y-axis is in log scale.

4. Sharing evaluation insights. In the last scenario, we will share with the attendees more in-depth insights that we gained while performing empirical evaluations on top of our generated graphs and query workloads, for both non-recursive and recursive (aka regular path) queries. In particular, our generator is the first one that supports *recursive queries*. The attendees will generate, visualize and import queries in the considered systems, to notice that even for very small graphs, the recursive queries are not efficiently handled, and that among all considered systems the Datalog engine has surprisingly the best performance. This scenario will point out important limitations of existing systems and highlight desirable directions of research for the graph data management community.

4. REFERENCES

[1] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified stress testing of RDF data management systems. In *ISWC*, pages 197–212, 2014.

[2] G. Aluç, M. T. Özsu, and K. Daudjee. Workload matters: Why RDF databases need a new design. *PVLDB*, 7(10):837–840, 2014.

[3] G. Bagan, A. Bonifati, R. Ciucanu, G. Fletcher, A. Lemay, and N. Advokaat. gMark: Controlling workload diversity in benchmarking graph databases, 2015. `http://arxiv.org/abs/1511.08386`.

[4] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The LDBC social network benchmark: Interactive workload. In *SIGMOD*, pages 619–630, 2015.

[5] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP2Bench: A SPARQL performance benchmark. In *ICDE*, pages 222–233, 2009.

[6] J. D. Ullman. *Principles of database and knowledge-base systems, Volumes I & II*. Computer Science Press, 1988, 1989.

[7] P. T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1):50–60, 2012.