gMark: Schema-Driven Generation of Graphs and Queries

Radu Ciucanu

Université Clermont Auvergne
Joint work with colleagues from Univ. Lille, Univ. Lyon, TU Eindhoven

JIRC 2017, Orléans
Why graph data?

Big **graph** data sets are ubiquitous
- social networks (e.g., LinkedIn, Facebook)
- scientific networks (e.g., Uniprot, PubChem)
- knowledge graphs (e.g., DBPedia)
- ...

Focus is on “**things**” and their **relationships**
Why graph databases?

Analytics on big graphs increasingly important
- role discovery in social networks
- identifying interesting patterns in biological networks
- finding important publications in a citation network
- ...

In response to these trends, the past decade has witnessed an explosion of graph data management solutions, e.g.,
- Graph databases such as Neo4j
- Graph analytics platforms such as GraphX
- Triple stores such as Virtuoso
- Datalog engines such as LogicBlox
Why graph database benchmarking?

Benchmark = data sets + query workloads

When a field has good benchmarks, we settle debates and the field makes rapid progress.

D. Patterson (CACM, 2012)

Motivated by success stories in relational and XML engineering e.g., TPC and XMark, it is clear that good benchmarks are needed for graph DBs
Graph database benchmarking

LDBC-SNB\textsuperscript{1} and WatDiv\textsuperscript{2} are current leaders in graph DBMS benchmarking

- **LDBC** is a fixed-schema and fixed-queries benchmark targeting focused stress-testing of query engineering choke-points
  - social network scenario

- **WatDiv** is a schema-driven workload-based benchmark targeting broad coverage of query features
  - default schema is products and users scenario

\textsuperscript{1}Erling, Averbuch, Larriba-Pey, Chafi, Gubichev, Prat, Pham, and Boncz: *The LDBC social network benchmark: Interactive workload*. SIGMOD’15.

\textsuperscript{2}Aluç, Hartig, Özsu, and Daudjee: *Diversified stress testing of RDF data management systems*. ISWC’14.
We present gMark, an open-source\textsuperscript{1} framework for generation of synthetic graphs and workloads.

Given a graph schema, gMark

\begin{itemize}
  \item generates synthetic instances of the schema (of desired size)
  \item generates sophisticated query workloads with targeted structure and runtime behavior (which holds for all instances of the schema)
\end{itemize}

\textsuperscript{1}https://github.com/graphMark/gmark
Why gMark?

We adopt successful aspects of the state of the art

Like WatDiv (and unlike LDBC), gMark is schema-driven,
- allowing finely tailored graph instances for specific application domains; and,
- allowing tightly controlled generation of query workloads.

Like LDBC (and unlike WatDiv), gMark supports focused stress-testing of query engineering choke-points, through fine control of query selectivities.
Why gMark?

Unlike both WatDiv and LDBC, gMark

- supports the generation of workloads containing recursive path queries, which are fundamental for graph analytics;

- performs selectivity estimation in a purely instance-independent schema-driven fashion.
  - hence, more scalable, more predictable, and easier to explain/understand
Overview of the gMark workflow

Graph configuration
- Size
- Node types
- Edge predicates
- Schema constraints
- Degree distributions

Query workload configuration
- Size
- Selectivity
- Recursion
- Shape
- Arity

Graph instance file (CSV)

SPARQL
openCypher
PostgreSQL
Datalog
1. Graph Generation

2. Query Generation

3. Scalability Study of Current Graph Databases

4. Evolving Graph Generation
gMark: Schema-Driven Generation of Graphs and Queries

1. Graph Generation

2. Query Generation

3. Scalability Study of Current Graph Databases

4. Evolving Graph Generation
gMark graph generation

Graph configuration
- Size
- Node types
- Edge predicates
- Schema constraints
- Degree distributions

gMark
Graph & query generator

Graph instance file (CSV)

Query workload configuration
- Size
- Selectivity
- Recursion
- Shape
- Arity

Query workload file (UCRPQs as XML)

Query translator
- SPARQL
- openCypher
- PostgreSQL
- Datalog
Graph configurations

The user can specify in the graph configuration (i.e., graph schema):

- **Size**: # of nodes
- **Node types**: finite set of node labels
  - e.g., author, citation, journal

- **Edge predicates**: finite set of edge labels
  - e.g., authoredBy, referencedBy

- **Schema constraints**: proportion of nodes/edges of given type
  - e.g., 20% of all nodes are authors

- **Degree distributions**: on the in- and out-degree of edge predicates (uniform, normal, zipfian)
  - e.g., the out-distribution of citation authoredBy author is Gaussian with parameters $\mu = 3, \sigma = 1$
Graph configurations: Uniprot schema

<table>
<thead>
<tr>
<th>Node type</th>
<th>Constr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>gene</td>
<td>35%</td>
</tr>
<tr>
<td>protein</td>
<td>31%</td>
</tr>
<tr>
<td>author</td>
<td>20%</td>
</tr>
<tr>
<td>citation</td>
<td>10%</td>
</tr>
<tr>
<td>organism</td>
<td>1%</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Edge predicate</th>
<th>Constr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>authoredBy</td>
<td>64%</td>
</tr>
<tr>
<td>encodedOn</td>
<td>6%</td>
</tr>
<tr>
<td>referencedBy</td>
<td>3%</td>
</tr>
<tr>
<td>occursIn</td>
<td>2%</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Node types**

**Edge predicates**

<table>
<thead>
<tr>
<th>source type predicate</th>
<th>target type</th>
<th>In-distr.</th>
<th>Out-distr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>citation</td>
<td>authoredBy</td>
<td>Zipfian</td>
<td>Gaussian</td>
</tr>
<tr>
<td>...</td>
<td>author</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**In- and out-degree distributions**
Schema-driven graph generation

We have established the intractability of the generation problem

**Theorem**

*Given a graph configuration $G$, deciding whether or not there exists a graph instance satisfying $G$ is NP-complete.*

Hence, gMark follows a ‘best-effort’ strategy in instance generation ($O(n)$), i.e., it attempts to achieve the exact values of the input parameters and relaxes them whenever this is not possible.
Schema-driven graph generation

We adapted the scenarios of popular use cases into meaningful gMark configurations, while also adding new gMark features:

- **Bib**: our default bibliographical use-case
- **LSN**: LDBC social network benchmark
- **WD**: WatDiv e-commerce benchmark
- **SP**: SP2Bench DBLP benchmark
Scalability of gMark graph generation

<table>
<thead>
<tr>
<th></th>
<th>100K</th>
<th>1M</th>
<th>10M</th>
<th>100M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bib</td>
<td>0m0.057s</td>
<td>0m0.638s</td>
<td>0m8.344s</td>
<td>1m28.725s</td>
</tr>
<tr>
<td>LSN</td>
<td>0m0.225s</td>
<td>0m1.451s</td>
<td>0m23.018s</td>
<td>3m11.318s</td>
</tr>
<tr>
<td>WD</td>
<td>0m2.163s</td>
<td>0m25.032s</td>
<td>4m10.988s</td>
<td>113m31.078s</td>
</tr>
<tr>
<td>SP</td>
<td>0m0.638s</td>
<td>0m7.048s</td>
<td>1m28.831s</td>
<td>15m23.542s</td>
</tr>
</tbody>
</table>

*Graph generation times, with varying graph sizes (# nodes)*

Generation time depends heavily on density of instances (e.g., WD has 100x number of edges than Bib)
gMark: Schema-Driven Generation of Graphs and Queries

1. Graph Generation

2. Query Generation

3. Scalability Study of Current Graph Databases

4. Evolving Graph Generation
gMark query generation

**Graph configuration**
- Size
- Node types
- Edge predicates
- Schema constraints
- Degree distributions

**Query workload configuration**
- Size
- Selectivity
- Recursion
- Shape
- Arity

---

**gMark**
Graph&query generator

**Query workload file**
(UCRPQs as XML)

**gMark**
Query translator

**Graph instance file**
(CSV)

- SPARQL
- openCypher
- PostgreSQL
- Datalog
A query language for graphs

UCRPQ: Unions of Conjunctions of Regular Path Queries
- Core constructs of the W3C’s SPARQL 1.1, Oracle’s PGQL, and Neo4j’s openCypher
- Well understood theoretical properties (e.g., polynomial data complexity)

UCRPQ includes recursive queries (via the Kleene star *), with applications in social networks, bioinformatics, etc.

gMark generates UCRPQ → the first synthetic workload generator to support recursive queries (and their translation in concrete syntaxes).
A query language for graphs

Example of UCRPQ

for each researcher, select all of the biological entities (i.e., genes and organisms) relevant to proteins studied in papers authored by people in the researcher’s coauthorship network

![Diagram showing query execution](image)
A query language for graphs

Example of UCRPQ

for each researcher, select all of the biological entities (i.e., genes and organisms) relevant to proteins studied in papers authored by people in the researcher’s coauthorship network

$(?x, ?z) \leftarrow (?x, (a\cdot a)^*, ?y), (?y, (a\cdot r\cdot e + a\cdot r\cdot o), ?z)$

(a=authoredBy, r=referencedBy, e=encodedOn, o=occursIn)

<table>
<thead>
<tr>
<th>#rules</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>#conjuncts</td>
<td>2</td>
</tr>
<tr>
<td>#disjuncts</td>
<td>1, 2</td>
</tr>
<tr>
<td>path length</td>
<td>2, 3, 3</td>
</tr>
</tbody>
</table>
Schema-driven workload generation

The user can specify in the query workload configuration:

- **Size**: #queries, #conjuncts/#disjuncts/path length per query

- **Selectivity**: constant, linear, quadratic.

- **Recursion**: probability to generate Kleene star above a conjunct.

- **Shape**: chain, star, cycle, star-chain.

- **Arity**: arbitrary (including 0 i.e., Boolean).

The graph configuration is also input to the query generator.
Selectivity estimation quality of gMark

- Given a binary query $Q$ and a graph $G$, we assume that $|Q(G)| = \mathcal{O}(\beta \times |\text{nodes}(G)|^{\alpha})$.

- $\alpha$ is the selectivity value (0–constant, 1–linear, 2–quadratic).

- Assigning selectivities required us to develop a selectivity algebra for instance-independent reasoning over query behavior.

- Experiments confirmed the assumption and the estimation quality.

<table>
<thead>
<tr>
<th></th>
<th>Constant</th>
<th>Linear</th>
<th>Quadratic</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSN</td>
<td>0.200 ± 0.417</td>
<td>1.189 ± 0.261</td>
<td>2.032 ± 0.059</td>
</tr>
<tr>
<td>Bib</td>
<td>0.003 ± 0.010</td>
<td>0.921 ± 0.122</td>
<td>1.405 ± 0.337</td>
</tr>
<tr>
<td>WD</td>
<td>0.016 ± 0.044</td>
<td>1.427 ± 0.392</td>
<td>2.004 ± 0.022</td>
</tr>
<tr>
<td>SP</td>
<td>0.074 ± 0.130</td>
<td>1.064 ± 0.034</td>
<td>2.034 ± 0.295</td>
</tr>
</tbody>
</table>
gMark query translator

**Graph configuration**
- Size
- Node types
- Edge predicates
- Schema constraints
- Degree distributions

**Query workload configuration**
- Size
- Selectivity
- Recursion
- Shape
- Arity

---

**gMark**
Graph&query generator

- Query workload file (UCRPQs as XML)
- Graph instance file (CSV)

---

**gMark**
Query translator

- SPARQL
- openCypher
- PostgreSQL
- Datalog
## Query translation

**UCRPQ:** \((?x, ?z) \leftarrow (?x, (a \cdot a)^*, ?y), (?y, (a \cdot r \cdot e + a \cdot r \cdot o), ?z)\)

<table>
<thead>
<tr>
<th>SPARQL</th>
<th>openCypher*</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PREFIX :</strong> <a href="http://example.org/gmark/">http://example.org/gmark/</a> SELECT DISTINCT ?x ?z WHERE { ?x (~a:a)* ?y . ?y ((~a:r:e)</td>
<td>(~a:r:o)) ?z .}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Datalog</th>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>g0(x,y)&lt;- edge(x1,a,x0),edge(x1,a,x2), x=x0,y=x2. g0(x,y)&lt;- g0(x,z),g0(z,y). g1(x,y)&lt;- edge(x1,a,x0),edge(x2,r,x1), edge(x2,e,x3),x=x0,y=x3. g1(x,y)&lt;- edge(x1,a,x0),edge(x2,r,x1), edge(x2,o,x3),x=x0,y=x3. query(x,z)&lt;- g0(x,y),g1(y,z).</td>
<td>WITH RECURSIVE c0(src, trg) AS ( SELECT edge.src, edge.src FROM edge UNION SELECT edge.trg, edge.trg FROM edge UNION SELECT s0.src, s0.trg FROM (SELECT trg as src, src as trg, ...........................................</td>
</tr>
</tbody>
</table>

* openCypher disallows Kleene star above concatenation or inverses.
Scalability of gMark workload generation

On a laptop, gMark generates workloads of one thousand queries for Bib in \( \sim 0.3s \); LSN and SP in \( \sim 1.5s \); and for the richer WD scenario in \( \sim 10s \).

Query translation of the thousand queries into all four supported syntaxes for each of the four scenarios requires \( \sim 0.1s \).
gMark: Schema-Driven Generation of Graphs and Queries

1. Graph Generation

2. Query Generation

3. Scalability Study of Current Graph Databases

4. Evolving Graph Generation
State-of-the-art graph DBMSs

We studied *query evaluation performance* of four mainstream graph DBMSs:

- **P**: PostgreSQL (SQL:1999 recursive views)
- **S**: a popular SPARQL query engine (SPARQL 1.1)
- **G**: a native graph database (openCypher)
- **D**: a modern Datalog engine (Datalog)
Scalability on non-recursive query workloads

Query execution times for diverse graph sizes and query workloads:
- **Len** (varying path lengths, 1 disjunct, 1 conjunct)
- **Dis** (multiple disjuncts, 1 conjunct)
- **Con** (multiple conjuncts and disjuncts)

<table>
<thead>
<tr>
<th>Scenario / System</th>
<th>2K</th>
<th>4K</th>
<th>8K</th>
<th>16K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant queries</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linear queries</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quadratic queries</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Constant queries**
- **Linear queries**
- **Quadratic queries**
Scalability on recursive query workloads

Query execution times for simple recursive queries on various small graph sizes (from 2K to 32K nodes):

![Graph showing query execution times](image-url)
gMark: Schema-Driven Generation of Graphs and Queries

1. Graph Generation

2. Query Generation

3. Scalability Study of Current Graph Databases

4. Evolving Graph Generation
Motivation

Graphs are naturally **evolving over time** e.g.,

- Nodes and edges have properties whose values change among consecutive snapshots
- Nodes and edges may exist only during specific time intervals

**Idea**: use gMark to generate schema-driven graphs and enrich them with time-evolving properties

\[
gMark + \text{time-evolving properties} = \text{EGG}^1
\]

---

1Open-source: https://github.com/karimalami7/EGG
EGG: Evolving Graph Generator

Evolving graph configuration
- # of snapshots
- Evolving properties (nodes and edges)
- Evolution constraints

Static graph configuration
- Size
- Node and edge types
- Occurrence constraints
- Degree distributions

EGG
Evolving graph generator

RDF annotated with temporal information

gMark
Static graph generator
Example

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>▶ e.g., 10M</td>
</tr>
<tr>
<td>Node types</td>
<td>▶ e.g., city, hotel</td>
</tr>
<tr>
<td>Edge predicates</td>
<td>▶ e.g., train, contains</td>
</tr>
<tr>
<td>Schema constraints</td>
<td>▶ e.g., 10% of all nodes are cities</td>
</tr>
<tr>
<td>Degree distributions</td>
<td>▶ e.g., the # of hotels in a city follows a Zipfian distribution</td>
</tr>
</tbody>
</table>

Evolving properties:

- city: weather, qAir
- hotel: star, availableRooms, hotelPrice
- train: trainPrice

Each graph snapshot corresponds to a day.
### Example

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>city</td>
<td>weather</td>
<td>unordered qualitative, has three possible values {sunny, cloudy, rainy} successors of sunny: sunny and cloudy.</td>
</tr>
<tr>
<td></td>
<td>qAir</td>
<td>ordered qualitative, has ten possible values from 1 to 10; can increment or decrement by 1 between two consecutive snapshots.</td>
</tr>
<tr>
<td>hotel</td>
<td>star</td>
<td>ordered qualitative, has values from 1 to 5, it changes every 365 snapshots with 1% probability, by one position at most</td>
</tr>
<tr>
<td></td>
<td>availableRooms</td>
<td>discrete quantitative, has values in ([1,100]); the offset is set to ([-15,15]).</td>
</tr>
</tbody>
</table>
|       | hotelPrice  | continuous quantitative, dependent on star for domain and on availableRooms for evolution e.g., for node \(x\) of type hotel:\
|       |             | if \(\text{star}(x) = 3\), then \(\text{hotelPrice}(x) \in [50,100]\)\
|       |             | if \(\text{availableRooms}(x) \uparrow\), then \(\text{hotelPrice}(x) \downarrow\)\
|       |             | if \(\text{availableRooms}(x) \downarrow\), then \(\text{hotelPrice}(x) \uparrow\).                                                   |
Summary of EGG contributions

- Linear-time generation algorithm

- Visualization module to emphasize the accuracy of EGG
Summary of EGG contributions

- **Storage format based on RDF named graphs** to decouple static and evolving parts of the graphs e.g.,
  
  \[
  \text{ns1:G31} \{ \text{<hotel:27> ns2:hasProperty <Property:availableRooms>} \} \\
  \text{ns1:snapshot9} \{ \text{ns1:G31 ns3:value "57" } \}
  \]

- **Evaluation of historical reachability queries**\(^1\) on top of EGG:
  - A baseline implementation in SPARQL on top of Apache Jena
  - Disjunctive-BFS: dynamic programming approach\(^1\)

---

Conclusions
Conclusions

- **gMark**
  - schema-driven graph and query-workload generator
  - finely controlled *query workload*-centered approach, featuring *instance-independent selectivity estimation*
  - translation to SPARQL, openCypher, SQL, Datalog
  - discovery of the poor performance of existing graph DBMS on evaluating a basic class of graph queries i.e., regular path queries

- **EGG**
  - *evolving graph generator* extending the gMark graphs with properties that evolve over time
  - storage format using *RDF named graphs* to reduce redundancy
  - easy to use to empirically evaluate evolving graph processing systems

---

1. https://github.com/graphMark/gmark
2. https://github.com/karimalami7/EGG
gMark & EGG papers

- Bagan, Bonifati, Ciucanu, Fletcher, Lemay, Advokaat
  *gMark: Schema-Driven Generation of Graphs and Queries*
  TKDE’17 full paper
  ICDE’17 extended abstract

- Bagan, Bonifati, Ciucanu, Fletcher, Lemay, Advokaat
  *Generating Flexible Workloads for Graph Databases*
  VLDB’16 demo

- Alami, Ciucanu, Mephu Nguifo
  *EGG: A Framework for Generating Evolving RDF Graphs*
  ISWC’17 demo

- Alami, Ciucanu, Mephu Nguifo
  *Synthetic Graph Generation from Finely-Tuned Temporal Constraints*
  TD-LSG @ PKDD/ECML’17