gMark: Controlling Workload Diversity in Benchmarking Graph Databases

Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurélien Lemay, Nicky Advokaat

Abstract—Massive graph data sets are pervasive in contemporary application domains. Hence, graph database systems are becoming increasingly important. In the study of these systems, it is vital that the research community has shared benchmarking solutions for the generation of database instances and query workloads having predictable and controllable properties. Similarly to TPC benchmarks for relational databases, benchmarks for graph databases have been important drivers for the Semantic Web and graph data management communities. In this paper, we present the design and engineering principles of gMark, a domain- and query language-independent graph benchmark exhibiting flexible schema and workload chokepoints. A core contribution of gMark is its ability to target and control the diversity of properties of both the generated graph instances and the generated query workloads coupled to these instances. A further novelty is the support of recursive regular path queries, a fundamental graph query paradigm. We illustrate the flexibility and practical usability of gMark by showcasing the framework's capabilities in generating high quality graphs and workloads, and its ability to encode user-defined schemas across a variety of application domains.

Index Terms—Graph databases, Selectivity estimation, Recursive queries, Benchmarking.

1 INTRODUCTION

The problem. It is well known that “when a field has good benchmarks, we settle debates and the field makes rapid progress” [1]. In this paper we study the problem of schema-driven generation of synthetic graph instances and corresponding query workloads for use in benchmarking graph database systems. Our study is motivated by the ubiquity of graph data in modern application domains, such as social and biological networks and geographic databases, and the consequent recent proliferation of graph data management solutions.

In response to these pressures, systems that can handle massive graph-structured data sets are under intense active research and development. These systems span from pure graph database systems to more focused knowledge representation systems. Native graph databases such as Neo4j [2] and Sparksee [3] propose their own declarative data model and query language, with particular attention to query optimization, and space and performance. In contrast to this trend of specialized systems, general-purpose systems such as LogicBlob [4] rely on declarative solutions that can cover a broader range of use cases. Furthermore, knowledge representation systems such as Virtuoso [5] implement the standard RDF graph data model and SPARQL query language to handle complex navigational and recursive queries on large-scale Semantic Web data.

Keeping pace with these developments, benchmarking graph data management systems has been a proliferating activity, which started within the Semantic Web community (e.g., [6], [7]) and recently within the database community through the extensive work of the LDBC Council [8]. The latter has been the first to propose a chokepoint-driven design of graph database benchmarks, which allows fine-grained control of chokepoints of queries and all aspects of the involved data. This lets the community focus on important features of query optimization and/or parallel processing, decreases the confusion and the incomparable results of the systems evaluated through the benchmarks, by relying on a fixed schema and a carefully designed set of benchmark queries.

Our solution. In this paper, we present a complementary approach to benchmarking in which the focus is not on individual queries but rather on whole query workloads. This approach relies on the control of diversity of both graph schemas and query workloads, which lets us vary the structural properties of data as well as tailor the generated queries to a particular domain or application. We believe that this approach can be useful in contexts in which multiple queries (i.e., those belonging to a query workload) need to be optimized together, for example, as in multi-query optimization, workload-driven database tuning, and streaming applications. We emphasize that a workload-centric approach primarily targets different benchmarking scenarios from the query-centric approach of current benchmarks such as those of the LDBC.
We realize this perspective with gMark, a system- and domain-independent graph benchmark framework that we present in this paper. gMark takes a schema-driven approach to the flexible and tightly-controlled generation of synthetic graph instances coupled with sophisticated query workloads. We provide gMark as open-source software[1] for use in the graph processing community.

To our knowledge, gMark is the first workload-centric graph benchmark to satisfy all of the following criteria: (i) being domain-independent, while at the same time targeting a rich variety of realistic domains; (ii) being extensible, capable of gauging both navigational and recursive query execution performance, while not relying on a fixed set of query templates that can be poorly tuned by the final consumers; (iii) being schema-driven, thus being capable of both targeting specific application domains and generating queries that adhere to the same schema used to generate graph instances; and, (iv) being data-diverse and query-diverse, thus being able to generate configurable data and configurable queries, along with a rich set of parameters such as bidirectional degree distributions and query selectivities, thus targeting and controlling the diversity of graph instances and their query workloads.

We further note that the core of query workload generation revolves around a novel method we propose here for selectivity estimation for graph queries, which is, to the best of our knowledge, the first of its kind. This method is of independent interest, applicable to other contexts such as query optimization and query inference on graphs.

1.1 gMark design principles

We next discuss the architectural design principles underlying gMark, in alignment with desiderata (i)-(iv) above. In order to support a broad range of systems and a broad range of domains, gMark expands our notion of graph benchmarking to include features and capabilities commonly found in graph query processing, graph analytics, and schema validation. The interplay between the following key features characterizes our system architecture.

Built-in support for schema definition. A major goal of gMark is to account for an array of fundamental user-defined schema constraints during graph generation. The graph generator can thus leverage an optional schema definition, called a graph configuration, which includes the enumeration of predicates (i.e., edge labels) and node types (i.e., node labels) occurring in the data, along with their proportions in generated instances. Default symbols for predicates and nodes can be redefined by means of aliases, thus accommodating the simulation of different application domains. Global distributions can be associated to predicates and node types, whereas in- and out-degree distributions can be defined on top of source(node)-to-target(node) schema constraints, optionally with multiplicities. Details about graph generation are given in Section[4]

Data and query workload diversity. Given a graph configuration, a subsequent challenge is that of exploiting it for query workload generation. Current approaches in benchmarking graph databases rely on the graph instances to generate queries of the workload with desired behavior. However, this approach is unfeasible for large and loosely structured networks. We argue that query workload generation must primarily rely on the graph configuration rather than on the generated graph instances, while still enforcing the desired behavior of the generated queries. Towards this, gMark supports a broad range of parameters in both the graph and query workload generator, the most notable of which are schema constraints with optional multiplicity in the former and query selectivities in the latter. To the best of our knowledge, none of the existing benchmarks for graph databases can support such configurable range of parameters. We discuss in Section[2] the relationship between gMark and contemporary graph benchmarking solutions. In Section[3] we show that it is an NP-complete problem to generate a graph (and a query workload) in the presence of the studied parameters. However, we later on empirically validate the high quality of the benchmarks generated by gMark.

Extensibility, language-, and system-independence. Another important design principle of gMark is extensibility, which involves the possibility of letting the user decide what she wants in terms of graph instances and query templates: size of the graphs; multiplicities and bidirectional degree distributions to simulate, e.g., real-world scale-free graphs; sizes of the queries; arity and shape of the queries (among star, chain, star-chain and cycles); and query selectivities and recursion can all be tuned to satisfy the user’s requirements. Additionally, gMark supports various practical output formats for the graphs and for the queries, including N-triples for data, and SQL, SPARQL, Datalog and openCypher as concrete query language syntaxes for query workloads. gMark is also easily extensible to support other output formats, and, as such, it exhibits language, and more generally, system independence.

Broad applicability. Finally, gMark broadly supports a wide range of applications domains. As an example, in this paper, we have encoded the graph schemas of three existing state-of-the-art benchmarks: the LDBC Social Network Benchmark[6], SP2Bench[7], and the Waterloo SPARQL Diversity Test Suite (WatDiv)[6]. More precisely, since gMark can be tuned to fit an arbitrary set of predicates and node types, we have been able to easily encode the ones for LDBC and SP2Bench (that rely on fixed schemas), and for WatDiv (that allows flexible schema specification and provides a default one). The expressible schema constraints in gMark and in these benchmarks are fairly not comparable as we explain in Section[6.1] and, in any case, the final goal of this study was rather to explain the expressiveness of our benchmark. Indeed, we have been able to easily adapt the scenarios of these benchmarks into meaningful gMark configurations while also adding some of our new features.

gMark v1.0. In its first release, gMark supports the full range of data and query features, and practical query syn- taxes discussed above. We depict an overview of the gMark workflow in Figure[1] and we describe its main components throughout the paper. Query selectivity tuning in this release is supported only on binary queries (i.e., queries of arity two). This should not be considered as a limitation since already selectivity tuning is a non-trivial problem for binary queries, and such queries already make interesting
practical cases of query benchmarking in graph databases (e.g., all regular path queries, which appear as “property paths” in SPARQL 1.1, are binary).

1.2 Contributions and organization

The goal of this paper is to present the design, engineering, and first empirical study of the gMark framework. We next outline our main contributions.

- We formalize the problems of graph generation and query workload generation, and show that they are intractable in general (Section 3).
- We provide an in-depth presentation of the gMark design principles, for the generation of both graphs (Section 4) and query workloads (Section 3). The most notable novel features are support for recursive queries and query selectivity estimation in the generated query workloads.
- We empirically show the capability of gMark to cover diverse graphs and query workloads, the accuracy of the estimated selectivities, and the scalability of the generator (Section 5).
- We present an in-depth experimental comparison of a representative selection of state-of-the-art graph query engines using gMark, which brings to light important limitations of current graph query processing engines, in particular w.r.t. recursive query processing (Section 7).

2 Related work

Benchmarking frameworks have played an important role in database systems research over the last decades, where efforts such as the TPC Benchmarks and XML benchmarking suites have been crucial in advancing the design and engineering of (semi)structured data management solutions [9–11]. With the increasing importance of loosely structured graph data in a wide range of application domains, there is a clear need for similar efforts in graph benchmarking. Indeed, a variety of synthetic benchmarking tools such as SP2Bench [7], LDBC [8], LUBM [12], BSBM [13], Grt [14] and WatDiv [6] have been developed in response to this need in the research community. Complementary to the gMark approach, application-driven derivation of graph configurations for benchmarking RDF databases has recently been studied in [15], [16]. Furthermore, extensive collections of real world network data sets such as SNAP [17] and KONECT [18] are now available as community resources.

All available graph benchmarking resources (i) rely on fixed graphs or instances of fixed graph schemas, or (ii) provide limited or no support for generating tailored query workloads to accompany graph instances. These aspects are difficult to jointly relax, especially in the context of loosely structured complex networks. Indeed, there is no community consensus on schema formalisms for graph data, an area which is still in an early stage of investigation [19]. Furthermore, constructing synthetic query workloads with given selectivity and structural features is extremely tricky [20]. As mentioned in the Introduction, current approaches such as WatDiv and LDBC perform selectivity estimation on generated graph instances, which becomes unfeasible when dealing with massive graphs and query workloads. In gMark, we address this challenge by generating tailored query workloads directly from the schema definition used for the generated graph instances and we set the query selectivity as one of the input parameters of such workloads. While BSBM and WatDiv do support a workload-centric approach to benchmarking, they do not provide this fine-grained level of control of query behavior. In general, we are not aware of any graph benchmarking solutions for controlling selectivity during query generation relying solely on graph schemas.

We further exemplify novel features of gMark w.r.t. SP2Bench and WatDiv in Section 3.1 while introducing our motivating example. Moreover, in Section 6.1 and, in deeper detail in Appendix A, we illustrate how gMark can be tuned to encode the schemas of LDBC, SP2Bench, and WatDiv.

To our knowledge, gMark is the first graph benchmark to generate workloads exhibiting recursive path queries. In particular, the queries generated by gMark are the so-called unions of conjunctive regular path queries [27–29]. This fundamental query language covers many graph queries which appear in practice. In particular, SPARQL 1.1 and openCypher have conjunctive regular path queries as their core constructs. They are also expressible in modern Datalog-like query languages [4] and in SQL:1999. As discussed
in the Introduction, gMark supports the output of query workloads in all these concrete query language syntaxes.

3 THE GENERATION PROBLEM

We start this section by intuitively introducing the generation problem via a real-world motivating example that also emphasizes some of the limitations of existing benchmarks (Section 3.1). Then, we formalize the benchmark generation problem i.e., generating a graph instance and a query workload on this instance according to a given set of constraints. More precisely, we formally define the problems of graph generation (Section 3.2) and query workload generation (Section 3.3), and we detail constraint parameters for both problems. We conclude the section by showing that the generation problem is intractable (Section 3.4).

3.1 Motivating example

Assume that a user wants to perform an extensive empirical evaluation of a new graph query processing algorithm that she designed. For this purpose, the user needs to efficiently generate: (i) graphs of different characteristics and sizes (to test the robustness and scalability of her algorithm), and (ii) query workloads sufficiently diverse to highlight strong or weak points of her new development. Additionally, our user would like to specify all parameters in a declarative way and to be able to simulate real-world scenarios.

For instance, the user would like 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. Moreover, the user would like 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 e.g., as in Figure 2(a) and 2(b). For instance, for graphs of arbitrary size, half of the nodes should be authors, 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 more or less constant).

Moreover, our user wants to specify real-world relationships between types and predicates via schema constraints e.g., 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.

Whereas specifying all aforementioned constraints as an input gMark graph configuration (cf. Figure 1) can be easily done via a few lines of XML, to the best of our knowledge there is no benchmark where they can be specified. For instance, in SP2Bench (which is also based on a similar bibliographical scenario), all constraints are hardcoded and the only parameter that a user can specify is the size of the graph, which makes it impossible for the user to finely tune schema-related characteristics of the graph. Moreover, in WatDiv [6], although the user can specify similar global constraints on the node types and the out-distributions, the absence of global constraints on the edge predicates and the absence of in-distributions entail important limitations, such as the absence of control on the selectivities of the queries of the generated query workloads.

In gMark we allow the user to finely tune the selectivities of the generated queries. For instance, the user can specify that she wants queries that, for any graph size, have constant, linear, or quadratic selectivity (we formally define these selectivity classes later on in the paper). To the best of our knowledge, no graph database benchmark supports such a feature. In particular, SP2Bench uses a fixed set of queries, while WatDiv can generate synthetic workloads, but without schema-driven selectivity control. As another remarkable difference to the state-of-the-art benchmarks, none of them supports recursive queries such as (authors-authors~)* which selects all pairs of researchers linked by a co-authorship path (by ~ we denote the predicate inverse and by * the transitive closure). As shown in the input gMark query workload configuration in Figure 1, the user can finely-tune e.g., the structure, size, selectivity of such queries.

3.2 Graph generation

gMark generates directed edge-labeled graphs and outputs them in formats that are compatible with the supported query languages. In this section, we formally define a graph configuration (cf. Figure 1), which is essentially a set of constraints that generated graph instances should satisfy. We start by giving a definition of the schema constraints which are the backbone of graph configurations.

Definition 3.1 A graph schema is a tuple $S = (\Sigma, \Theta, T, \eta)$ where $\Sigma$ is a finite alphabet of predicates, $\Theta$ is a finite set of types such that each node of the generated graph is associated with exactly one type, $T$ is a set of constraints on $\Sigma$ and $\Theta$ associating to each predicate and type either a proportion of its occurrences or a fixed constant value, and $\eta$ is a partial function associating to a triple consisting of a pair of input and output types $T_1, T_2$ in $\Theta$ and a symbol $a$ in $\Sigma$, a pair $(D_{in}, D_{out})$ of in- and out-degree distributions.
Predicates correspond to edge labels, and in the remainder we use the two terms interchangeably. A degree distribution is a probability distribution, among which gMark currently supports uniform, Gaussian (also known as normal), and Zipfian distributions. For each distribution, the user can specify the relevant parameters (i.e., min and max for uniform, $\mu$ and $\sigma$ for Gaussian, and $s$ for Zipfian). If the user wants to specify only the in- or the out-distribution, she can mark the other one as nonspecified. We discuss in Section 6.1 some examples where finely tuning such distributions is indeed meaningful from a practical point of view. Notice that the parameters for Example 3.3 of the graph.

For each triple $T_1, T_2, a$ have to be consistent in order to guarantee the compatibility of the number of generated ingoing and outgoing edges. We discuss the details of this consistency check in Section 4.

**Definition 3.2** A graph configuration is a tuple $G = (n, S)$, where $n$ is the number of nodes of the graph and $S$ is the schema of the graph.

**Example 3.3** Take a graph configuration $G = (n, S)$ s.t.:
- The graph should have $n = 5$ nodes.
- The graph should satisfy the schema $S = (\Sigma, \Theta, T, \eta)$, where $\Sigma = \{a, b\}$, $\Theta = \{T_1, T_2, T_3\}$, $T$ is defined as $T(T_1) = 60\%$, $T(T_2) = 20\%$ and $T(T_3) = 1$ and $\eta$ is defined as follows (we report only some constraints):

\[
\begin{align*}
\eta(T_1, T_1, a) &= (g, z), & \eta(T_1, T_2, b) &= (u, g), \\
\eta(T_2, T_2, b) &= (g, ns), & \eta(T_2, T_3, b) &= (ns, u),
\end{align*}
\]

where by $u, g, z$, and $ns$ we denote uniform, Gaussian, Zipfian, and non-specified distributions, respectively. For instance, the graph in Figure 3 can be generated by using this graph configuration. Although a much larger graph is needed to observe the actual distributions, we refer to Section 5 for further examples also handling distributions.

### 3.3 Query workload generation

We next formally define query workload configurations (cf. Figure 1). Towards this, we first outline the query language supported by gMark. As motivated in Section 2 we focus on generating unions of conjunctions of regular path queries (UCRPQ). This fundamental query language covers many queries which appear in practice, including the core constructs of SPARQL 1.1 queries, Neo4j’s Cypher queries, and many Datalog-based encodings [27]–[29].

Recall that $\Sigma$ is a finite alphabet (cf. Definition 3.1) and let $\Sigma^+ = \{a, a^- \mid a \in \Sigma\}$, where $a^-$ denotes the inverse of the 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

\[
(?)v \leftarrow (?x_1, r_1, ?y_1), \ldots, (?x_n, r_n, ?y_n)
\]

where: for each $1 \leq i \leq n$, it is the case that $?x_i, ?y_i \in V$; $?v$ is a vector of zero or more of these variables, the length of which is called the arity of the rule; and, for each $1 \leq i \leq n$, it is the case that $r_i$ is a regular expression over $\Sigma^+$ 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 $\Sigma^+$. We refer to the right-hand side of a query rule as the body of the query rule, each subgoal $(?x_i, r_i, ?y_i)$ of the body as a conjunct, and to the left-hand side as the head of the query rule.

A query $Q \in$ UCRPQ is a finite non-empty set of query rules, each of the same arity. The semantics $Q(G)$ of evaluating $Q$ on a given graph $G$ (having edge labels in $\Sigma$) is the standard one following that of unions of conjunctive Datalog queries [27]–[29], assuming standard set-oriented semantics. In summary, a query is basically a collection of query rules, each rule having several conjuncts, each conjunct having several disjuncts whose paths have a certain length.

**Example 3.4** Take the following UCRPQ query:

\[
(\bar{x}, \bar{y}, \bar{z}) \leftarrow (\bar{x}, (a \cdot b + c)^*, \bar{y}), (\bar{y}, a, \bar{w}), (\bar{w}, b^-, \bar{z}) \\
(\bar{x}, \bar{y}, \bar{z}) \leftarrow (\bar{x}, (a \cdot b + c)^*, \bar{y}), (\bar{y}, a, \bar{z})
\]

This query selects nodes $x, y, z$ such that one can navigate between $x$ and $y$ with a path in the language of $(a \cdot b + c)^*$, and moreover, can navigate between $y$ and $z$ with a path in the language of $a \cdot b^* + a$. This query consists of two rules consisting of three conjuncts and two conjuncts, resp. The conjuncts of the form $(?x, (a \cdot b + c)^*, {?y})$ have two disjuncts (of length 2 and 1, resp.) and all other conjuncts have only one disjunct (of length 1).

We define query size as a tuple

\[
t = ([f_{\text{min}}, f_{\text{max}}], [c_{\text{min}}, c_{\text{max}}], [d_{\text{min}}, d_{\text{max}}], [l_{\text{min}}, l_{\text{max}}])
\]

providing intervals of minimal and maximal values for the number of rules, conjuncts, disjuncts, and length of the paths in the query, resp., that generated queries should have. For example, the query from Example 3.4 has size $([2, 2], [2, 3], [1, 2], [1, 2])$. In gMark, users can specify minimal and maximal values for all of these parameters; in turn, the query generation algorithm can assign values that range in these intervals. For simplicity of presentation, we assume in the remainder that a query consists of only one rule.

**Definition 3.5** A query workload configuration is a tuple $Q = (G, \#q, ar, f, e, p_r, t)$ where $G$ is a graph configuration, $\#q$ is the number of queries in the workload (defined on all instances of $G$), $ar$ is the arity constraint, $f$ is the shape constraint, $e$ is the selectivity of the queries in the workload, $p_r$ is the probability of recursion, and $t$ is the query size.

Notice that in addition to the graph configuration $G$ (cf. Section 3.2), the user can specify several other constraints. First, $ar$ is the range of allowed arities for the queries in the workload. For instance, the query from Example 3.4 has arity 3. We also support Boolean queries (arity 0). The shape constraint $f$ contains the supported query shapes.
(among which chain, star, cycle, and star-chain are currently supported in gMark) and the user can specify which among them she would like to have in the generated query workload. Similarly, the selectivity constraint $e$ contains the desired selectivity classes, among which we support constant, linear and quadratic (cf. Section 5.1). The user can further specify the probability to have the multiplicity $*$ above a disjunct, reflected by the parameter $p_i$.

We finally point out that gMark is able to translate the generated UCRPQ in four concrete syntaxes (cf. Figure 1): SPARQL, openCypher, PostgreSQL, and Datalog.

3.4 Intractability of the generation problem

In this section, we prove the intractability of the problems of graph and query workload generation. First, we prove that the graph generation problem is intractable.

Theorem 3.6 Given a graph configuration $G$, deciding whether there exists a graph satisfying $G$ is NP-complete.

Prior to presenting the proof of Theorem 3.6 we would like to introduce some standard macros for encoding pairs of in- and out-distributions. Precisely, we use:

- $\eta_i$: for non-specified in-degree distribution and uniform out-degree distribution with min=max=1. In other words, $\eta_i(T_i, T_2, a) = 1$ means that from a node of type $T_i$ there is precisely one outgoing $a$-labeled edge to a node of type $T_2$, and that in a node of type $T_2$ we can have an arbitrary number of incoming $a$-labeled edges from nodes of type $T_i$.
- $\eta_0$: for non-specified in-degree distribution and uniform out-degree distribution with min=0 and max=1.
- $\eta_0$: for non-specified in-degree distribution and uniform out-degree distribution with min=max=0.

Proof We show the NP-hardness by reduction from the SAT$_{1\min=3}$ problem, known to be NP-complete [30]. Take a 3CNF formula $\varphi = C_1 \land \ldots \land C_k$ over variables $x_1, \ldots, x_n$. We construct a schema $S_0 = (\Sigma_0, \Theta_0, T_0, \eta_0)$ and a subsequent graph configuration $G_0 = (\eta_0, S_0)$ as follows:

1. $\Sigma_0$: The graph should have $2 \times n + k + 1$ nodes.
2. $\Theta_0$: There should be $3 \times n + k$ symbols (predicates) in the alphabet: $\Sigma_0 = \{c_1, \ldots, c_{2n}, b_1, \ldots, b_n, t_1, \ldots, t_n, f_n\}$.
3. $T_0$: There should be precisely one node of type $A$ in the graph, which can be expressed as $T_0(A) = 1$. Additionally, $T_0(B_1) = \ldots = T_0(B_n) = T_0(C_1) = \ldots = T_0(C_k) = 1$.
4. $\eta_0$: There should be 1/(2 * n + k + 1) of the graph nodes.

- $\eta_0$ such that:
  - $\eta_0(A, T_1, t_1) = \ldots = \eta_0(A, T_n, t_n) = 1$.
  - $\eta_0(A, F_1, f_1) = \ldots = \eta_0(A, F_n, f_n) = 1$.
  - $\eta_0(T_i, C_1, c_1) = \ldots = \eta_0(T_i, C_m, c_m) = 1$, for every $i \in \{1, \ldots, n\}$, where $c_1, \ldots, c_m$ correspond to the clauses in which the variable $x_i$ appears in a positive literal.
  - $\eta_0(F_1, C_1, c_1) = \ldots = \eta_0(F_1, C_m, c_m) = \eta_0(F_1, B_i, b_i) = 1$, for every $i \in \{1, \ldots, n\}$, where $c_1, \ldots, c_m$ correspond to the clauses in which the variable $x_i$ appears in a negative literal.

For the only if part, take a valuation that satisfies exactly one literal of each clause and construct a graph that encodes this valuation, starting from a node of type $A$. For example, for the above formula $\varphi_0$ and the valuation such that $x_1$ and $x_2$ are true, and $x_3$ and $x_4$ are false, construct the graph from Figure 4. Since we choose exactly one among $T_1$ and $F_1$ for every $i \in \{1, \ldots, n\}$ we have exactly one $B_i$. Moreover, since exactly one literal of each clause is satisfied we have exactly one $C_i$ (for $i \in \{1, \ldots, k\}$). Thus, the constraints in $T_0$ are satisfied. As for the number of nodes, we have 2 * n (because there is exactly one valuation of each variable and we also have its corresponding $B_i + k$ (because of the $k$ clauses) + 1 (the node of type $A$). Consequently, the constructed graph satisfies all the constraints from the configuration.

For the if part, take a graph satisfying the constraints. Since it satisfies the constraints from $T_0$, the graph should have one node $A$, one $B_i$ (for $i \in \{1, \ldots, n\}$) and one $C_i$ (for $i \in \{1, \ldots, k\}$). Since the total size of the graph is 2 * n + k + 1 nodes and seen how we can reach $B_i$’s and $C_i$’s based on the schema, we infer that the other $n$ nodes correspond to nodes of type $T_i$ or $F_i$ (encoding a valuation of the variable $x_i$). Since we have precisely one $B_i$, we infer that each variable has exactly one valuation and since we have precisely one $C_i$, we infer that in each clause there is exactly one literal that is satisfied. This means that the formula $\varphi$ is in SAT$_{1\min=3}$.

To show the membership of the problem to NP, we point out that a non-deterministic Turing machine has to guess a graph having as many nodes as the constraint from the configuration. The size of such a graph is thus polynomial in the size of the input and testing whether it satisfies the schema can be easily done in polynomial time. $\square$
As a natural consequence of Theorem 3.6, we have that the query workload generation problem is also intractable. That means that some parameters of the query workload cannot be fulfilled and it is not possible to test this efficiently. That is why gMark follows a “best effort” strategy in the generation: it tries to achieve the exact values of the parameters and relaxes them whenever this is not possible.

**Corollary 3.7** Given a query workload configuration $Q$, deciding whether there exists a query workload satisfying $Q$ is NP-complete.

**Proof** Recall that a query workload configuration $Q$ is a tuple $(G, \#q, ar, f, e, p_r, t)$. Since the graph configuration $G$ is part of the input of $Q$, we can take for $G$ precisely the same encoding of SAT$_{1-in-3}$ as in the proof of Theorem 3.6, and then arbitrary values for the other constraints in $Q$. Then, both $if$ and $only if$ parts for the NP-hardness follow precisely as in the proof of Theorem 3.6. As for the membership of the problem to NP, notice that a non-deterministic Turing machine has to guess a set of queries satisfying the given constraints, in particular of size $t$ hence polynomial in the size of the input. Moreover, notice that deciding whether a given query workload satisfies the input constraints can be also easily decided in polynomial time.

Despite the intractability of the generation problems and the gMark “best effort” approach, we would like to already point out that the gMark graph and query generator leads to highly accurate results, as we detail with our experiments in Section 6.2, where we report on the selectivity estimation of the generated queries over the generated graphs.

### 4 Graph Generation

The gMark graph generation algorithm is conceptually quite straightforward. Given graph schema $S = (\Sigma, \Theta, T, \eta)$, the generation algorithm considers the in- and out-degree distributions $D_{in}, D_{out}$ for each triple $(T_1, T_2, a)$ in the domain of $\eta$. If both distributions are Gaussian, the number of edges that has to be created is computed as the average number of edges per node, multiplied by the number of nodes. Then, every edge is drawn by picking uniformly at random its origin and destination nodes from the set of nodes of that type, i.e., $T_1$ or $T_2$. The label $a$ of the corresponding triple $(T_1, T_2, a)$ is assigned to such an edge.

If only one distribution, e.g., the out-degree distribution, is Gaussian, the algorithm first creates a vector containing all nodes of the input type $T_1$ a number of times which is drawn uniformly at random according to the input distribution $D_{in}$. The algorithm shuffles this vector and then creates all edges one at a time by simply reading the shuffled vector for the input type $T_1$, and performs a random draw of the output node $T_2$.

If none of the distribution is Gaussian (i.e., they are either uniform or Zipfian), the algorithm essentially creates two vectors with the same technique mentioned above and shuffles them. It then produces edges one at a time by reading both vectors simultaneously. Note that a difference of size may occur between the two vectors, in which case the generation procedure ends when it has finished reading one of the two vectors.

### 5 Query Generation

A core innovation of gMark is that graph instances and query workloads are both generated from graph schemas. This allows the tight coupling of queries to instances while still also supporting fine-grained control of the diversity of query workloads. As discussed in Section 3.3, query generation in gMark is guided by the same input schema used for graph generation, which makes the queries of the workload pertinent to graph instances. Furthermore, along with the query shape and size, users can specify query selectivity.

In this section, we first formalize our notion of selectivity. Our focus here is on selectivity classes for binary queries, a natural and broad class in the context of graph queries, strictly containing the regular path queries (i.e., property paths in SPARQL 1.1). In the balance of the section, we present the gMark algorithm for generating queries of specified structure and selectivity.

#### 5.1 Selectivity classes for binary queries

The selectivity of a query $Q$ on a graph $G$ is the number of results returned by the evaluation of $Q$ on $G$, i.e., $|Q(G)|$. This number depends on both the topology and the actual size of the graph instance, given that instances of different sizes can be generated upon the same input schema.

Given a schema $S$ and all graphs $G$ satisfying $S$, we assume that the value $|Q(G)|$ behaves asymptotically as a function of the form $|Q(G)| = \beta |G|^{\alpha}$, where $\alpha$ and $\beta$ are real constants. We say that the above value $\alpha$ is the selectivity value of $Q$ w.r.t. $S$, denoted $\alpha_S(Q)$ or simply $\alpha(Q)$ when it does not lead to ambiguity. Thus, the selectivity value of a query is by definition bounded by the query arity. Since in this section we focus on binary queries, we consider selectivity values such that $0 \leq \alpha(Q) \leq 2$.

**Classes of binary queries.** Using the above formalization, we identify three practical query classes, depending on whether $\alpha(Q)$ is closer to 0, 1, or 2:

- **Constant queries** (for which $\alpha(Q) \approx 0$) select a number of results that does not grow (or barely grows) with the graph size. For instance, a query selecting pairs (country, language) is constant if the graphs follow a realistic schema specifying that the numbers of countries and languages do not grow with the graph size, and hence the number of query results is more or less constant.

- **Linear queries** (for which $\alpha(Q) \approx 1$) select a number of results that grows at a rate close to the growth of the number of nodes in the graph instances. For example, a query selecting pairs (language, user) is linear if the schema specifies that the number of users grows with the graph, whereas the number of languages is more or less constant. Another example of a linear query is (user, address) if we assume that the schema specifies that each user has precisely one address and the number of users grows linearly with the graph.

- **Quadratic queries** (for which $\alpha(Q) \approx 2$) select a number of results that grows at a rate close to the growth of the square of the number of nodes in the graph instances. For example, the transitive closure of the knows predicate in

---

2. Our experimental study duly confirms this assumption.
a social network is quadratic because a realistic schema should specify that this predicate follows a power-law (e.g., Zipfian) in- and out-distribution. Thus, the query results contains Cartesian products of subsets of users that know and are known by some hub users of the social network.

**Estimating the selectivity value.** We propose a solution for estimating the selectivity value \( \alpha(Q) \) of a given query \( Q \), for all graphs satisfaying a given schema \( S \). This basically means to compute a function that associates to \( Q \) a value \( \hat{\alpha}(Q) \in \{0, 1, 2\} \). This value can be made more precise as follows: for a pair of node types \( A \) and \( B \), \( \hat{\alpha}_{A,B}(Q) \) is the estimated selectivity of \( Q \) restricted to pairs \( (x,y) \) where \( x \) is of type \( A \) and \( y \) of type \( B \). Then, the overall estimated selectivity value of \( Q \) is \( \hat{\alpha}(Q) = \max(\hat{\alpha}_{A,B}(Q)) \).

To compute these values, we define an algebra based on what we call selectivity classes. First, for each node type \( A \) within the input schema \( S \), we denote Type\( (A) = N \) if \( A \) grows with the graph size and Type\( (A) = 1 \) if it does not. In the graph schema, Type\( (A) = 1 \) if \( T(A) \) is a fixed value and Type\( (A) = N \) if \( T(A) \) is a proportional value.

For each query \( Q \) and each pair of node types \( A \) and \( B \), the selectivity class of \( Q \) for \( A,B \), denoted sel\( _{A,B}(Q) \) is a triple \((t_A, o, t_B)\) such that \( t_A = \text{Type}(A), t_B = \text{Type}(B) \) and \( o \in \{=, <, >, \diamond, \times\} \) is an operator between types.

We summarize these algebraic operations in Table 1, which should be read as follows: an operation from the first column denotes that for every graph \( G \) satisfying a schema \( S \), for every pair of nodes \( (n_1, n_2) \in Q(G) \), it is the case that \( |\{n \mid (n_1, n) \in Q(G)\}| \) and \( |\{n \mid (n, n_2) \in Q(G)\}| \), resp., are or are not bounded (by some constants), as indicated in the second and third columns, resp. The last column \( \alpha(Q) \) is particularly useful to distinguish between the last two operations \( \diamond \) and \( \times \).

We next intuitively explain the above operations and we illustrate them via examples:

- \( = \) is the simplest operation and occurs either (i) between constant types e.g., (country, language) as illustrated for constant queries, or (ii) for some linear queries such as the query defined by the empty regular expression \( \varepsilon \) that returns precisely as many results as the nodes in the graph.

- \( < \) characterizes queries where either (i) the out-degree distribution is Zipfian, such as (language, user) as illustrated for linear queries, or (ii) the source node type \( A \) has Type\( (A) = 1 \) and the target node type \( B \) has Type\( (B) = N \). Then, the definition of \( > \) is symmetric to \( < \).

- \( \times \) corresponds to queries performing a Cartesian product between two node sets (both growing with the graph), for example the transitive closure of the knows predicate that we used above to illustrate the quadratic queries. Intuitively, the \( \times \) is the result of a \( > \) followed by a \( < \).

**Selectivity classes for regular path queries.** Recall from Section 3.3 that gMark workloads consist of queries expressed as UCRFPQ’s. Hence, we need to compute the selectivity values for regular path queries, which involve regular expressions. First, for a query \( Q \) defined by the regular expression \( \varepsilon \) (the empty word), for each type \( A \), we have that \( \text{sel}_{A}(Q) = \text{Type}(A), \text{Type}(A) \). When \( Q \) is defined by a single edge label \( a \in \Sigma^+ \), we obtain \( \text{sel}_{A,B}(Q) \) directly from the distribution of the \( a \)-labeled edges from \( A \) to \( B \), as defined above and obtained from the schema.

**Example 5.1** Consider the schema given in Example 3.3. First, we assign selectivity classes to the types of the schema, thus Type\( (T_1) = \text{Type}(T_2) = N \) whereas Type\( (T_3) = 1 \). From the schema, we compute the following values:

- \( \text{sel}_{T_1,T_1}(a) = (N, <, N) \) and \( \text{sel}_{T_1,T_1}(a^-) = (N, >, N) \) (because of the Zipfian out-distribution that moreover implies a Zipfian in-distribution for the inverse, and Type\( (T_1) = N) \).

- \( \text{sel}_{T_1,T_2}(b) = (N, =, N) \) and \( \text{sel}_{T_2,T_1}(b^-) = (N, =, N) \) (because of non-Zipfian in- and out-distributions, and moreover, both Type\( (T_1) = \text{Type}(T_2) = N \)).

- \( \text{sel}_{T_2,T_2}(b) = (N, =, N) \) and \( \text{sel}_{T_2,T_2}(b^-) = (N, =, N) \) (same reasoning as for the previous bullet).

- \( \text{sel}_{T_1,T_3}(b) = (N, >, 1) \) and \( \text{sel}_{T_3,T_1}(b^-) = (1, <, N) \) (because of non-Zipfian in- and out-distributions, and moreover, Type\( (T_2) = N \) and Type\( (T_3) = 1 \)).

Let a query \( Q \) be defined by the regular expression \( p_1 + p_2 \) where \( p_1 \) and \( p_2 \) are two regular expressions that define queries \( Q_1 \) and \( Q_2 \), respectively. For every pair of node types \( A,B \), such that \( \text{sel}_{A,B}(Q_1) = (t_A, o_1,t_B) \) and \( \text{sel}_{A,B}(Q_2) = (t_A, o_2,t_B) \) then \( \text{sel}_{A,B}(Q) = (t_A, o_1 + o_2,t_B) \) where \( o_1 + o_2 \) is defined by the table in Figure 5(a).

![Table 1: Algebraic operations between types.](image)

![Figure 5: Algebra for selectivity classes.](image)
(1, \circ, 1) are not permitted, which makes \((1, =, 1), (1, <, N)\)
and \((N, >, 1)\) the only permitted triples that contain a 1. However, in the
computation of the algebraic expression, we could still obtain triples \((1, \times, 1)\)
and \((1, \circ, 1)\), that we should replace with \((1, =, 1)\) if the case occurs.

Finally, the estimated selectivity value of a query is obtained directly
from its class. If the obtained selectivity class of a query \(Q\) is \((1, =, 1)\), then \(\hat{\alpha}(Q) = 0\), if we obtain \((N, \times, N)\), then \(\hat{\alpha}(Q) = 2\), and \(\hat{\alpha}(Q) = 1\) for all other cases.

5.2 Query generation algorithm

We are now ready to present the gMark query generation algorithm. For simplicity of presentation, we focus on binary chain queries and discuss extensions (the majority of them already implemented in gMark) in Section 5.3.

**Algorithm 1 Query workload generation algorithm.**

**Input:** A query workload configuration \(Q\)

**Output:** A set of queries \(Q\)

1: \(\text{let } Q = \emptyset\)

// Step 1: Prepare Selectivity Graphs
2: \(\text{let } G_S = \text{get\_schema\_graph}(S)\)
3: \(\text{let } D = \text{get\_distance\_matrix}(G_S)\)
4: \(\text{let } G_{sel} = \text{get\_selectivity\_graph}(D)\)
5: \(\text{for } i \in \{1, \ldots, \#Q\}\)

// Step 2: Generate query skeleton
6: \(\text{let } Q = \text{get\_query\_skeleton}(t)\)

// Step 3: Get selectivity class for each predicate
7: \(T = \text{assign\_types}(Q, S, G_{sel})\)

// Step 4: Generate paths of predicates
8: \(\text{for } P \in \text{conj}(Q)\)
9: \(\text{let } P = \text{get\_predicate\_template}(t)\)
10: \(P = \text{fill\_paths}(G_S, T(P))\)
11: \(\text{Add } Q \text{ to } \emptyset\)

The query generation algorithm outputs queries satisfying the parameters given in the *query workload configuration* \(Q\) (cf. Section 5.3). The generation procedure follows four main steps. (1) First, it prepares all necessary ingredients to compute the selectivity, namely a graph that blends the schema with selectivity triples (1a), a distance matrix (1b) and a selectivity graph (1c). Then, it yields a *query skeleton* (2) based on the number of conjuncts, (3) computes the selectivity class for each conjunct and then (4) instantiates each conjunct by (4a) fixing the number of disjuncts and the length of paths of each predicate and (4b) selecting the edge labels for those paths in order to achieve the desired selectivity. We detail each of these steps below.

**Step 1a - Generate the schema graph.** We first use the schema \(S\) to build the so-called schema graph \(G_S\). Each node in \(G_S\) is a pair given by a node type of the schema and a selectivity triple (cf. Section 5.1) associated to that type. More formally, the set of nodes of the schema graph \(G_S\), denoted \(\text{SelType}(S)\), consists of tuples \((T, (t_1, o, Type(T)))\), where (i) \(T\) is a node type from \(\Theta\) and (ii) \((t_1, o, Type(T))\) is a selectivity triple in the set of all possible selectivity triples \{(1, =, 1), (1, <, N), \ldots\}. The edges of \(G_S\) are labeled with symbols in \(\Sigma^+\).

**Fig. 6. A snippet of the schema graph for our running example.**

The goal of the schema graph \(G_S\) is to indicate how a chain query whose path ends with a type \(T\) of selectivity triple \((t_1, o, Type(T))\) changes when its path is extended with an edge in \(\Sigma^+\). Formally, given a node \((T, (t_1, o, Type(T)))\) and a label (or label inverse) \(a \in \Sigma^+\) such that the schema allows an \(a\)-labeled edge between \(T\) and a node type \(T'\), if according to our algebra we have \((t_1, o, Type(T)) \cdot \text{sel}_{T,T'}(a) = (t_1, o', Type(T'))\), then in the schema graph \(G_S\) there is an edge \(((T, (t_1, o, Type(T))), a, (T', (t_1, o', Type(T'))).\)

**Example 5.2** Recall Examples [3,3] and [5.1] We illustrate a snippet of the corresponding schema graph in Figure 5. For instance, the node \((T_1, (N, =, N))\) is due to the fact that \(Type(T_1) = N\) and recall from Section 5.1 that for a given type \(A\) we have \(\text{sel}_{A, A}(\varepsilon) = (\text{Type}(A), \text{Type}(A))\). Moreover, our schema allows an \(a\)-labeled edge between two nodes of type \(T_1\) following a Zipfian out-degree distribution, hence its selectivity triple is \((N, <, N)\), which explains the node \((T_1, (N, <, N))\) in Figure 5. Additionally, there is an \(a\)-labeled edge in our schema graph between the nodes \((T_1, (N, =, N))\) and \((T_1, (N, <, N))\) because in our algebra \((N, =, N) \cdot (N, <, N) = (N, <, N)\).

**Step 1b - Compute the distance matrix.** The distance matrix \(D\) establishes for each pair \(n, n' \in \text{SelType}(S)\) the length \(D(n, n')\) of the shortest path between \(n\) and \(n'\) in \(G_S\).

**Step 1c - Generate the selectivity graph.** At this step, we build a selectivity graph \(G_{sel}\) i.e., an unlabeled directed graph whose nodes are \(\text{SelType}(S)\). An edge exists between two nodes \(n\) and \(n'\) if there exists a path between \(n\) and \(n'\) of length in \([l_{\text{min}}, l_{\text{max}}]\) (recall that this interval is specified as part of the query size in the workload configuration \(Q\)). This is approximated by checking if \(D(n, n') < l_{\text{max}}\).

**Example 5.3** We illustrate the selectivity graph for our running example in Figure 4. As an example, there exists a path between \(T_1, (N, =, N)\) and \(T_2, (N, \times, N)\) (for instance, the path \(b \cdot b \cdot b^*\)), whose length is less than \(l_{\text{max}} = 4\). However, there does not exist such a path between \(T_2, (N, \times, N)\) and \(T_1, (N, =, N)\), thus there is no edge between those nodes.

**Step 2 - Construct query skeletons.** A query skeleton consists of a set of variables \(V\), a set of predicates \(\Sigma^+\) and a set of conjuncts of the form \((x_1, P, x_2)\) where \(x_i \in V\) and \(P\) is constructed with predicates in \(\Sigma^+\). The function \(\text{get\_query\_skeleton}(t)\) creates a query skeleton according to the number of conjuncts specified in the configuration. At this stage, notice that the predicates are merely uninstantiated placeholders.
In the case of chain queries, if the number of conjuncts \( c_{\text{min}} = c_{\text{max}} = 3 \), we shall obtain a query skeleton as follows: \( Q = (x_1, P_1, x_2) \land (x_2, P_2, x_3) \land (x_3, P_3, x_4) \).

**Step 3 - Assign selectivity types.** We build a function \( T \) that associates to each predicate \( P \) of the query skeleton a selectivity type \((T_1, \text{Type}(T_1), o, \text{Type}(T_2)), T_2\)\), where \( T_1 \) and \( T_2 \) are schema types, and \( o \) is a selectivity operator. This function exhibits two properties. First, the input and output type of each selectivity operator should be consistent with the types of \( T_1 \), \( T_2 \) and secondly, selectivity values for each type should be selected in a way to guarantee that the global selectivity class for the query is the expected one. To achieve this, we randomly choose a path on the selectivity graph \( G_{\text{sel}} \). For chain queries, this basically means that we want to find a path between a node with selectivity tuple \( (?=,=?) \) (where by “?” we denote any type) to a node with \((T_1, o, T_2)\) yielding one of the desired selectivities. The length of this path should be consistent with configuration.

**Example 5.4** In our running example, assuming that we look for a linear chain query with 3 conjuncts, we can instantiate the function \( T \) as follows:

\[
T(P_1) = (T_1, (N, =, N), T_1), \\
T(P_2) = (T_1, (N, >, N), T_2), \\
T(P_3) = (T_2, (N, =, N), T_2)
\]

We can then compute the selectivity of the concatenation, which is \((N, =, N) \cdot (N, >, N) \cdot (N, =, N) = (N, >, N)\), which corresponds to a linear query.

Note that drawing uniformly at random paths of a certain length in \( G_{\text{sel}} \) can be done efficiently with a two-step algorithm: first, each node \( n \) is associated with a function \( nb\_path(n, i) \) that gives the number of paths of length \( i \) that can be generated starting from \( n \). For instance, for a quadratic query, a node \( n_1 : (A, (N, \times, N)) \) has \( nb\_path(n_1, 0) = 1 \) whereas a node \( n_2 : (B, (1 = 1)) \) has \( nb\_path(n_2, 0) = 0 \). Other values are obtained by a saturation algorithm: to generate a path of length \( l \), the algorithm picks a starting node with a random draw weighted by \( nb\_path(n, l) \), and then picks the label of an outgoing edge to a node \( n' \) with a random draw weighted by \( nb\_path(n', l - 1) \), and so on until all the nodes are saturated.

**Step 4 - Instantiate paths.** In this final step, we instantiate predicates of the conjuncts constructed in the above steps with paths that satisfy the query workload parameters. First, we build for each predicate a path skeleton that satisfies the constraints concerning the number of disjuncts and the length of the paths in the query size specified by \( Q \), as illustrated by the following example.

**Example 5.5** To continue our running example, with a number of disjuncts in the range \([d_{\text{min}}, d_{\text{max}}] = [3, 5]\) and path length in the range \([l_{\text{min}}, l_{\text{max}}] = [2, 4]\), we may have a path skeleton as follows: \( P_1 = X_1 \cdot X_2 + X_3 \cdot X_4 \cdot X_5 + X_6 \cdot X_7 \) (i.e., three disjuncts having path length from 2 to 3).

Once a path skeleton is computed, we need to find actual edge labels for each path, as in Step 3 by randomly choosing paths in \( G_{\text{sel}} \). For the sake of conciseness, we omit the details and we illustrate it on the following example.

**Example 5.6** Let us consider \( P_1 = X_1 \cdot X_2 + X_3 \cdot X_4 \cdot X_5 + X_6 \cdot X_7 \), with \( T(P_1) = (T_1, (N, =, N), T_1) \). The query workload generation algorithm may obtain a path instantiation of this kind: \( X_1 = b \) and \( X_2 = b^- \) as the path \( b \cdot b^- \) can go from \( T_1 \) to \( T_2 \) and from \( T_2 \) to \( T_1 \) via a concatenation of two paths with edge label \( b \).

### 5.3 Extensions to recursive queries, and more diverse query shapes and arities

In Section 5.2 we presented our query workload generation algorithm for binary queries having a chain shape. We discuss here the extensions to recursion in the query predicates, along with other query shapes and arities. All these extensions are all implemented in gMark v1.0, whereas the selectivity estimation is guaranteed only for binary queries.

First, we generate recursive queries by adding Kleene stars to chain queries as follows. After building a query skeleton, we choose the number of predicates on which there is recursion, as well as their position. The selectivity triple for every other predicate is then chosen by ignoring the recursive predicate. Recursive predicates inherit the input and the output types of their neighbor nodes in the chain, with the selectivity operator ‘*’.

Second, chain queries are the basic ingredients of all the other query shapes and dealing with them implies an extension of the basic case for chain queries. Other query shapes are essentially combinations of different chain queries: cycle queries can be considered as two chain queries that share the same selectivity and the same endpoint variables, star queries are a combination of chains that have the same starting variable, and chain-star queries are obtained by combining a chain query with a star query.

Third, to generate queries of arbitrary arity, we start by generating queries satisfying the shape, size and recursion constraints, and then we randomly pick a set of projection variables such that their number is consistent with the arity constraint. gMark v1.0 already supports the generation of queries of arbitrary arity (including Boolean i.e., arity 0).

The fact that the selectivity estimation is for the moment guaranteed only for binary queries should not be considered as a limitation since it already involved the development of novel sophisticated machinery (as shown in Section 5.2). Moreover, such queries already make interesting practical cases for graph database benchmarking (e.g., all regular path queries, known as “property paths” in SPARQL 1.1, are binary) and point out important limitations of existing graph database engines, as we highlight in Section 6.

### 6 Empirical evaluation of gMark

In this section, we empirically evaluate gMark w.r.t. two important aspects: the capability to encode the application
TABLE 2
\(\alpha(Q)\) averaged across constant, linear, and quadratic queries (with standard deviation), with varying graph sizes, data, and query diversity.

<table>
<thead>
<tr>
<th></th>
<th>Constant</th>
<th>Linear</th>
<th>Quadratic</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSN-Len</td>
<td>0.280±0.417</td>
<td>1.189±0.261</td>
<td>2.032±0.059</td>
</tr>
<tr>
<td>LSN-Dis</td>
<td>0.182±0.364</td>
<td>1.325±0.318</td>
<td>2.046±0.074</td>
</tr>
<tr>
<td>LSN-Con</td>
<td>0.190±0.301</td>
<td>1.244±0.326</td>
<td>2.017±0.032</td>
</tr>
<tr>
<td>LSN-Rec</td>
<td>0.196±0.409</td>
<td>1.099±0.492</td>
<td>1.564±0.889</td>
</tr>
<tr>
<td>Bib-Len</td>
<td>0.003±0.011</td>
<td>0.951±0.122</td>
<td>1.405±0.331</td>
</tr>
<tr>
<td>Bib-Dis</td>
<td>0.000±0.000</td>
<td>0.995±0.012</td>
<td>1.607±0.261</td>
</tr>
<tr>
<td>Bib-Con</td>
<td>0.023±0.029</td>
<td>0.986±0.112</td>
<td>1.409±0.296</td>
</tr>
<tr>
<td>Bib-Rec</td>
<td>0.100±0.316</td>
<td>0.982±0.073</td>
<td>1.493±0.335</td>
</tr>
<tr>
<td>WD-Len</td>
<td>0.016±0.044</td>
<td>1.427±0.392</td>
<td>2.004±0.022</td>
</tr>
<tr>
<td>WD-Dis</td>
<td>0.000±0.022</td>
<td>1.412±0.380</td>
<td>1.999±0.014</td>
</tr>
<tr>
<td>WD-Con</td>
<td>-0.010±0.020</td>
<td>1.540±0.495</td>
<td>1.730±0.708</td>
</tr>
<tr>
<td>WD-Rec</td>
<td>0.587±0.830</td>
<td>1.976±0.012</td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td>0.074±0.130</td>
<td>1.064±0.634</td>
<td>2.034±0.295</td>
</tr>
</tbody>
</table>

domain of existing benchmarks (Section 6.1), and its quality in terms of both accuracy of the estimated selectivities and scalability of the generator (Section 6.2).

Environment. All experiments reported in this section and in Section 7 were run on an Intel Core i7 920, with 6GB RAM, and running Ubuntu 14.04 64bit.

6.1 Coverage of practical graph scenarios

In our experiments, we relied on four use cases: the default gMark use case, and three gMark encodings of the schemas of existing state-of-the-art benchmarks [6]–[8], which were possible because gMark can be easily tuned to fit an arbitrary set of predicates, node types, and schema constraints.

- **Bib** is our default scenario, describing the bibliographical database introduced in Section 5.1 as our motivating example. It represents a baseline, illustrating the main gMark features, in particular all types of degree distributions.
- **LSN** is our gMark encoding of the fixed schema provided with the LDBC Social Network Benchmark [8], which simulates user activity in a social network.
- **SP** is our gMark encoding of the fixed DBLP-based [31] schema provided with SP2Bench [7].
- **WD** is our gMark encoding of the default schema provided with the Waterloo SPARQL Diversity Test Suite (WatDiv) [6], which differs from LDBC and SP2Bench, and is similar to gMark in the sense that it also supports user-defined schemas via a so-called dataset description language. The default schema that we encoded is about users and products.

The difference between gMark and the aforementioned three benchmarks resides in the kind of expressive schema constraints, which are incomparable. Nonetheless, we have been able to encode their key characteristics, while often adding some of our new features. We detail in Appendix A the expressiveness differences between gMark and existing benchmarks, and our encoding choices of their schemas.

6.2 Quality and scalability

Selectivity estimation quality study. Our first set of experiments focuses on understanding the quality of the selectivity estimation performed using the algebra presented in Section 5.1. We used four types of workloads in which we stress-test several diverse queries with incrementally varied query size: Len generates queries with varying path lengths, no disjuncts, no conjuncts, and no recursion; Dis generates queries with disjuncts, no conjuncts and no recursion; Con generates queries with conjuncts and disjuncts and no recursion; Rec generates queries with recursion (Kleene-stars).

Finally, the last row (SP) of Table 2 shows the estimated \(\alpha\)-values for SP2Bench [7] on a set of queries following our gMark encoding of the original set of SP2Bench queries. In the remainder of our study in Section 7, we disregard SP since this use case does not bring more insights than the query sets generated for the other use cases.

To further illustrate the precision of our estimated values, we report in Figure 8 the estimated selectivities (\(|E|\)) along with the theoretical selectivities (\(|Q|\)) for constant (\(Q_1\)), linear (\(Q_2\)), and quadratic (\(Q_3\)) queries on the Bib use case. We observe that for the classes of queries of increasing expressiveness the number of results is generally higher for quadratic queries, while it is linearly and constantly varying for the other queries, as expected. We also observe that the two curves representing the estimated selectivities and the theoretical ones closely overlap in all the cases. Finally, notice that the above experiments are considering chain queries only. The results on other shapes and/or use cases are similar and omitted for conciseness.

We conclude from this study that the schema-driven gMark selectivity estimation framework generates consistently high quality estimates, across all selectivity classes, across a broad spectrum of diverse data sets and queries.

Scalability study. Our second set of experiments is devoted to measuring the time taken by the graph generator of gMark, while varying the size of the data and the size of the query workload. To gauge the robustness of our


7 Evaluation of Query Engines

We next turn to an empirical evaluation of a representative selection of currently available graph query processing engines using gMark. Our goal here is both to demonstrate the new capabilities in benchmarking introduced by gMark, and to pinpoint limitations and areas for further improvement in current graph query processing solutions.

7.1 Design of experiments

Systems. The database systems (and their supported query languages) that we consider in our study are:

- **G**: a native graph database (openCypher [32])
- **S**: a popular SPARQL query engine (SPARQL 1.1 [33])
- **P**: PostgreSQL v9.3.9 (SQL:1999 recursive views [35])
- **D**: a modern Datalog engine (Datalog [36])

For obvious reasons, we obfuscate the names of the three commercial systems employed in our study.

3. For obvious reasons, we obfuscate the names of the three commercial systems employed in our study.

4. We use the standard translation of UCRPQ's into recursive views, implemented using linear recursion [29], [34].

For the sake of fairness, we used the default configuration for each system i.e., without special purpose optimizations.

Query languages. Recall that the queries generated by gMark are UCRPQ's. We provide in Appendix B a translation of an example UCRPQ into each of the above concrete syntaxes. We note that not all systems support the full expressive power of UCRPQ's. In particular, arbitrary UCRPQ's can be expressed in SPARQL, SQL, and Datalog, while openCypher supports only those UCRPQ's having no occurrences of converse or concatenation under Kleene star. In our results regarding recursive queries, some of the generated benchmark queries do indeed exhibit converse and/or concatenation in a recursive conjunct. In these cases, the corresponding openCypher query has only the non-converse symbol and/or the first symbol in a concatenation of symbols, respectively. Furthermore, while all other languages adopt the classical homomorphic semantics for conjunctive queries [36], openCypher adopts an isomorphic semantics. For these two reasons, openCypher queries often have answer sets that differ from that of their counterparts in the other languages, which should be kept in mind while evaluating experimental results pertaining to system G.

Measurements. We generate and execute query workloads on a variety of graph configurations. We execute and measure the runtime of each query six times. The first one is a “cold” run that we exclude from the computation of the average; from the remaining five “warm” runs we drop the fastest and slowest and then report the average of the remaining three execution times. Between the execution of each query, we close and reopen the database to clear any caching effects. We consider the following parameters in generating workloads: selectivity classes of the tested queries (i.e., constant, linear, quadratic) and size (amounting to 30 queries for each workload). We consider the following parameters in generating graph instances: size (from 2K to 16K nodes) and use case i.e., Bib, LSN, and WD (cf. Section 6.1).

We make three remarks here. (i) Despite being small, the considered graph sizes were already sufficient to illustrate interesting behavior and distinctions between the studied systems. Indeed, as we discuss in Section 7.2 already on these graphs we observed that many simple queries fail on a majority of the systems. Even for those queries that succeed, the evaluation times are often very high e.g., hundreds or...
even thousands of seconds already on instances of these sizes. (ii) To ensure a fair comparison of all systems and to avoid measuring the time to print the query results, we added to all queries the aggregate `count(distinct(?v))`, where `?v` is the (binary) vector of output variables. We recall that `distinct` is also necessary for our analysis since the algebra relies on the elimination of duplicates (cf. Section 5). (iii) In this study all queries are chain shaped, as this is the basic query shape from which the others are constructed and hence are sufficient for illustrating the relative performance of current graph query processing engines. Finally, even though in the presentation of the results, we focus on the default use case Bib, we observed comparable trends for the other two use cases, which we omit for the sake of conciseness.

7.2 Results of experiments

Nonrecursive queries. In our first experiment, we focus on the nonrecursive queries i.e., query workloads Len, Dis, Con (cf. Section 6.2). We summarize the results in Figure 9. The goal of this study is to observe how the different systems react to these varied workloads. Figure 9(a) shows the query execution times averaged across the 10 constant queries of each workload. Out of the 10 averages obtained by the 5 warm runs of each query, we computed once again the overall average, by discarding two out of the 10 averages that have the farthest standard deviation with respect to this overall average. This allows to capture the cases in which some of the systems fail or give outlier results. In Figure 9(a), we observe that P reacts better than S, G, and D to query workload diversity, by exhibiting lower query evaluation times on all instance sizes. This behavior is confirmed in the case of linear queries, as shown in Figure 9(b) for the Con query workload on all the sizes and for the Len and Dis query workloads for smaller sizes only i.e., 2K and 4K. For larger instance sizes i.e., 8K and 16K, the behavior is reverted in favor of S for linear queries. Then, as shown in Figure 9(c) for quadratic queries, S continues on this trend by beating P, G, and D. We also observe from all query execution times reported in Figure 9 that the times taken by constant and linear are of the same order of magnitude, whereas quadratic queries, as expected, typically exhibit an order of magnitude slowdown. There is only one system (D) for which the differences of the behavior along the sets of linear and quadratic queries are blurred.

We conclude from this study that gMark allows us to generate interesting queries and diverse query workloads that already on small graph instances stress-test state-of-the-art systems, and highlight particular strengths and weaknesses in processing graph queries. As a general observation, we can further conclude that the straightforward standard implementation of UCRPQ’s in PostgreSQL typically shows superior performance across a broad class of queries (i.e. constant and linear) to that of existing dedicated systems.

Recursive queries. Our second experiment is devoted to recursive queries, generated by query workloads containing Kleene stars. Unfortunately, all tested systems either failed on the majority of these queries or had to be manually terminated after unexpectedly long running times. For these reasons, it is difficult to draw a clear conclusion on recursive queries. Therefore, we performed a small case analysis: we considered two recursive queries of constant and quadratic selectivity, respectively, for which we could collect results for at least one of the four systems. We report the results for both queries in Table 4. The first query has constant selectivity. P was quite slow at evaluating it on small instances and starts failing on graphs of 8K nodes. S was able to answer this query only on the smallest graph size (2K). Failed in all cases and always returned empty results (due to its different query semantics, as discussed in Section 7.1). The only system for which we could measure the evaluation time for all sizes was D, which also turned to be the most efficient one. The second query has quadratic selectivity and only D was able to evaluate it. We conclude from this study that only D is currently able to deal with recursive queries.

8 Concluding remarks

We presented gMark, the first graph benchmark solution that satisfies the key criteria of being domain-independent, extensible, schema-driven, and highly configurable also in terms of the expected query selectivity of a given workload. The latter is a novel contribution on its own and is applicable to other independent benchmarks and problems, such as query optimization and query inference on graphs. Furthermore, gMark is the first benchmark to generate workloads exhibiting recursive path queries, which are central to graph querying. Our in-depth empirical study demonstrated both the quality and practicality of gMark. Moreover, our experiments highlighted important limitations in the query processing capabilities of current state-of-the-art graph processing engines, already on small graph instances and on both recursive and non-recursive queries.

With the first release of gMark, we plan to now align our work with international benchmarking bodies such as the LDBC Council. Looking ahead to the second release, there are many directions for further investigation (e.g., extending the selectivity estimation to n-ary queries).

### References

2. [http://neo4j.com](http://neo4j.com).

5. For instance, we could envision the query workload generation in gMark applied to real graph datasets on top of which a schema extraction tool has been run beforehand.
**Fig. 9.** Summary of query execution times for diverse query workloads (Len, Dis, Con) and various graph sizes under Postgres (P) and three commercial systems: a SPARQL query engine (S), a native graph database (G), and a modern Datalog engine (D).

<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>

[10] [http://konect.uni-koblenz.de](http://konect.uni-koblenz.de)


[17] [http://konect.uni-koblenz.de](http://konect.uni-koblenz.de)


APPENDIX

A Details on use cases

In this section, we provide further details on the gMark encoding of the LDBC [8], SP2Bench [7] and WatDiv [6] benchmarks (introduced in Section 6.1) and discuss the adequacy of gMark for modeling them as graph configurations.

LDBC. The LDBC Social Network Benchmark [8] has a fixed schema that simulates user activity in a social network. Its schema consists of 11 entities (such as Persons, Tags, Forums, Messages, Likes, Organizations, Places, etc.) and 20 relations (such as knows, studyAt, workAt, likes, hasMember, etc.). There are three main differences between LDBC and gMark: (i) LDBC relies on the aforementioned fixed schema, while gMark can be configured to fit an arbitrary schema, (ii) LDBC supports subtyping (e.g., City, Country, and Continent are all subtypes of the type Place) whereas this kind of feature is out of the scope of gMark, and (iii) LDBC encodes the associations between the different entities via standard UML notation (such as one-to-many or many-to-many associations) whereas gMark can finely tune these associations with in- and out-degree distributions. Consequently, since the LDBC social network benchmark and gMark are incomparable, we had to make some choices when modeling LDBC in our framework. More precisely, regarding (i) and (ii) above, we have not specified in gMark the types that have subtypes but rather have specified directly the subtypes (e.g., we do not consider the type Place but we consider instead all of its subtypes City, Country, and Continent). More importantly, regarding (iii), we used our fine-grained support for encoding distributions between entities to replace generic associations with more natural distributions between them. For example, we encoded the relation knows between two persons with a Zipfian on both the in- and out-distributions (since in a real-world network there are indeed some authorities that know and are known by many others, while the majority of people know and are known by few others), and we encoded workAt with a Zipfian in-distribution and a uniform out-distribution (since intuitively a person works for a small number of companies, while there are few companies with a lot of employees and more companies with few employees). To sum up, typical real-world associations such as knows and workAt, encoded simply as many-to-many relationships in LDBC, can be modeled with finer grain by using the gMark bidirectional distribution control.

SP2Bench. The SP2Bench [7] benchmark has a fixed DBLP-based schema [31]. The only parameter that the user can control in the graph generator is the graph size (given either as number of tuples or number of years with publications), and the generator hardcodes a large number of constraints to make the generated graphs resemble as much as possible with the real DBLP. On the other hand, the user can declaratively specify in gMark much more parameters than only the size of the graph. Due to its generality, gMark cannot encode very specific characteristics such making the parameters μ/σ for the Gaussian distribution depend on the value of the publication year as SP2Bench does. Nonetheless, gMark easily captures the essential properties, such as the power-law distributions (found in the citation system or the distribution of papers among authors). Moreover, SP2Bench contains constraints like x% of the nodes of type A have an attribute B, that we encode in gMark as follows: we specify how many A’s and B’s we want in the graph, and the distributions between A and B. From the query workload point of view, SP2Bench was designed specifically to cover as much as possible from the SPARQL operators, whereas we are independent from the concrete query language. At the moment when SP2Bench was designed, the SPARQL specification did not include yet the property paths, whereas gMark covers such a feature. Moreover, the SP2Bench designers relied on the knowledge of the schema to be able to manually design a few meaningful queries with different characteristics, whereas gMark automatically generates a controlled diverse workload for an arbitrary schema.

WatDiv. The Waterloo SPARQL Diversity Test Suite (WatDiv) [6] differs from LDBC and is similar to ours in the sense that it also supports user-defined schemas via a so-called dataset description language. However, the kind of schema constraints expressible in WatDiv and gMark differ. Indeed, WatDiv supports only local constraints for the predicates (that are however different from ours), while gMark supports global constraints on both types and predicates. More precisely, the local constraints in WatDiv specify for instance that 30% of the products have a content rating (we used a similar encoding as explained above for SP2Bench for such constraints), whereas gMark can encode that there is a Zipfian in- and out-distribution between product and rating. Additionally, gMark can specify global constraints also on the predicates (e.g., 40% of the edges should be labeled friendOf) whereas WatDiv does not. Another difference is that WatDiv relies solely on RDF while our model is general enough to support any graph encoding. However, since they rely on RDF, they also make use of subtyping similar to LDBC (e.g., a product can be an album, a movie, a concert, etc.). For the goal of our study, we encoded the standard WatDiv test dataset schema (based on products, retailers, users, etc.) while using only the subtypes instead of general types and adapting their local constraints to our semantics.

B Examples of query translations

In this section, we give an example query in UCRPQ along with its translations into all four practical query languages supported by gMark, namely SPARQL, Datalog, openCypher, and SQL. The query that we discuss here contains one query rule, and consists of both non-recursive and recursive conjuncts, along with disjuncts and inverses.

We first report the query in the syntax used throughout the paper:

\[(?x, ?y, ?z) \leftarrow (?x, (a^{-} + c^{+}), ?y, (b^{+} - d^{+} \cdot d^{-} \cdot e), ?z).\]

Then, we show its gMark translation into SPARQL 1.1:

```
PREFIX : <http://example.org/gmark/>
SELECT DISTINCT ?x ?y ?z
WHERE {?x ((^:p0) | (:p2))^ ?y .
?y ((^:p1) | (:p3/:^:p3/:p4)) ?z .}
```

Next, we show its gMark translation into Datalog:

```
g0(x,y) <- edge(x1, 0, x0), x = x0, y = x1.
```
\( g_0(x, y) \leftarrow \text{edge}(x_0, 2, x_1), x = x_0, y = x_1. \)
\( g_0(x, x) \leftarrow \text{edge}(x, r, y). \)
\( g_0(y, y) \leftarrow \text{edge}(x, r, y). \)
\( g_0(x, y) \leftarrow g_0(x, z), g_0(z, y). \)
\( g_1(x, y) \leftarrow \text{edge}(x_1, 1, x_0), x = x_0, y = x_1. \)
\( g_1(x, y) \leftarrow \text{edge}(x_0, 3, x_1), \text{edge}(x_2, 3, x_1), \text{edge}(x_2, 4, x_3), x = x_0, y = x_3. \)

query \((x, y, z) \leftarrow g_0(x, y), g_1(y, z).\)

Next, we show its gMark translation into openCypher:

\[
\text{MATCH (x)-[}:p0|p2*:]-\rightarrow(y), (y)<-[}:p1]-\rightarrow(z)
\text{RETURN DISTINCT x, y, z}
\]
\[
\text{UNION}
\text{MATCH (x)-[}:p0|p2*:]-\rightarrow(y),
(y)-[}:p3]-\rightarrow()-[}:p3]-\rightarrow()-[}:p4]-\rightarrow(z)
\text{RETURN DISTINCT x, y, z;}
\]

Finally, we show its gMark translation into SQL:

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, label FROM edge) as s0 WHERE s0.label = 0
    UNION
    SELECT s0.src, s0.trg FROM edge s0 WHERE s0.label = 2 )
c1(src, trg) AS (
    SELECT src, trg FROM c0
    UNION
    SELECT head.src, tail.trg FROM c0 as head, c1 as tail WHERE head.trg = tail.src),
c2(src, trg) AS (
    SELECT s0.src, s0.trg FROM (SELECT trg as src, src as trg, label FROM edge) as s0 WHERE s0.label = 1
    UNION
    SELECT s0.src, s2.trg FROM edge s0,
    (SELECT trg as src, src as trg, label FROM edge) as s1, edge s2
    WHERE s0.trg = s1.src AND s1.trg = s2.src
    AND s0.label = 3 AND s1.label = 3
    AND s2.label = 4 )
SELECT DISTINCT c1.src, c2.src, c2.trg FROM c1, c2 WHERE c2.src = c1.trg;