LSQB: A Large-Scale Subgraph Query Benchmark

Amine Mhedhbi
University of Waterloo
amine.mhedhbi@uwaterloo.ca

Matteo Lissandrini
Aalborg University
matteo@cs.aau.dk

Laurens Kuiper
CWI Amsterdam
laurens.kuiper@cwi.nl

Jack Waudby
Newcastle University
j.waudby2@newcastle.ac.uk

Gábor Szárnyas
CWI Amsterdam
gabor.szarnyas@cwi.nl

ABSTRACT

We introduce *LSQB*, a new large-scale subgraph query benchmark. LSQB tests the performance of database management systems on an important class of subgraph queries overlooked by existing benchmarks. Matching a labelled structural graph pattern, referred to as subgraph matching, is the focus of LSQB. In relational terms, the benchmark tests DBMSs' join performance as a choke-point since subgraph matching is equivalent to multi-way joins between base Vertex and base Edge tables on `ID` attributes. The benchmark focuses on read-heavy workloads by relying on *global queries* which have been ignored by prior benchmarks. Global queries, also referred to as unseeded queries, are a type of queries that are only constrained by labels on the query vertices and edges. LSQB contains a total of nine queries and leverages the LDBC social network data generator for scalability. The benchmark gained both academic and industrial interest and is used internally by 5+ different vendors.

ACM Reference Format:
Amine Mhedhbi, Matteo Lissandrini, Laurens Kuiper, Jack Waudby, and Gábor Szárnyas. 2021. LSQB: A Large-Scale Subgraph Query Benchmark. In *4th Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA'21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3461837.3464516

1 INTRODUCTION

Subgraph queries are a fundamental class of queries for applications where graph patterns reveal valuable information. For example, biologists and sociologists identify communities in large networks by finding dense subgraphs [17], Twitter searches for diamonds in their users follower network to provide "whom-to-follow" recommendations [21], and Alibaba detects fraudulent activities by finding cycles [43].

In the property graph data model [33], vertices represent entities, edges represent relationships, and arbitrary key-value pairs represent properties on vertices and edges. Contemporary *graph DBMSs* (GDBMSs), which use the property graph data model, support subgraph queries.

GRADES-NDA'21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8477-3/21/06.
https://doi.org/10.1145/3461837.3464516

As observed in prior work [1, 3, 32], a subgraph matching query $Q(\mathrm{V}_Q, \mathrm{E}_Q)$, which enumerates instances of Q in an input graph $G(\mathrm{V}, \mathrm{E})$, is equivalent to a `select-project-join` query containing multi-way joins between base Vertex and base Edge tables. Therefore, provided a mapping from the graph schema to the relational schema, *relational DBMSs* (RDBMSs) also support subgraph queries.

1.1 Subgraph Query Workloads Overview

Read-heavy analytical applications containing subgraph queries keep gaining popularity [47]. Such applications obtain graph data from social networks, web crawls, or from the integration of multiple datasets stored in transactional RDBMSs. These application workloads differ from classical relational workloads in the abundance of three operations:

(1) *Many-to-many joins* are prevalent since highly connected graph data contains a very large number of many-to-many relationships. Even a small number of input tuples to such joins leads to an explosion in the size of intermediate and output results.
(2) *Cyclic joins* are fundamental for applications such as social network recommendation and fraud detection.
(3) *Long acyclic joins* are employed for path-finding use cases possibly having great depths.

1.2 The Need for New Benchmarks

While GDBMSs, e.g., Neo4j [55], TigerGraph [12], and GraphflowDB [25] are specifically optimized for subgraph query workloads, RDBMSs, e.g., Db2 [53], SAP HANA [46], and Umbra [34] are expanding their query processing and optimization techniques to perform better on these workloads. An example of an effective query processing technique for subgraph queries in GDBMSs and RDBMSs are the newly developed worst-case optimal joins (WCOJs) [35] supported by GraphflowDB [32], SAP HANA [56], and Umbra [19].

With any newly developed query processing or optimization technique, we turn to benchmarks to quantify the promised improvements and inevitable tradeoffs. Micro-benchmarks demonstrate that the design decisions are indeed responsible for successfully achieving runtime improvements [31]. End-to-end macro-benchmarks showcase the overall system performance, identify any cross-cutting issues, and help avoid regressions as systems evolve [24, 27].

The database community sees existing benchmarks for subgraph queries as dispersed, ad-hoc, and lacking a principled suite of micro-benchmarks [39, 48]. Specifically, prior work studying subgraph query evaluation present highly divergent methodologies. In particular, current benchmarks do not contain a common suite of subgraph queries, rarely compare on the same datasets, and arbitrarily use directed and undirected subgraph queries [1, 3, 19, 28, 32, 44]. Therefore, a well thought-out unified benchmark focusing on the common operations mentioned in Section 1.1 is necessary. The benchmark needs to identify common tradeoffs by varying query structures and provide a consistent view of how techniques compare over time.

1.3 The Landscape of Existing Benchmarks

1.3.1 Existing Benchmarks Overview. We classify subgraph query workload benchmarks within a quadrant along two axes: (i) *query complexity* denoting from a high level the number of joins and portion of data accessed; and (ii) *"structuredness" of data* [13] defining what assumptions can be made a-priori as to what type(s) of data the system is going to handle. The quadrant is shown in Figure 1 and contains existing major benchmarks such as WatDiv [2], SNB [4], LinkBench [6], YCSB [11], SPB [27], and GMB [31].

Not accounting for "structuredness", most of the existing benchmarks consist of seeded queries, i.e., queries start from a given vertex or set of vertices. The quadrant orders the benchmarks by *read-intensity* from left to right with the SNB BI benchmark being the most read-intensive.

1.3.2 An Overview of the LDBC SNB. The *Linked Data Benchmark Council* (LDBC) developed a large scale macro-benchmark called the *Social Network Benchmark* (SNB) [4]. LDBC developed the benchmark following a choke-point driven methodology leading to two query sets of interest: Interactive (Int) and Business Intelligence (BI). LDBC's SNB is an effective benchmark since it is: (i) portable between the relational and property graph data models; (ii) scalable and representative of real workloads; and (iii) easy to understand due to domain simplicity. The benchmark still has many shortcomings however. SNB is rather intimidating as it requires too many features and is hard to implement for research prototypes. Recent academic research tends to use varied subsets of SNB queries focusing on a subset of choke-points [20, 28]. SNB is quite similar to relational workloads, as it is heavily inspired by TPC-H [14], and emphasizes complex filtering and aggregation operations.

1.4 Contributions

We introduce a new labelled subgraph query benchmark (LSQB), to make progress on facilitating meaningful performance testing while correcting for the shortcomings of prior work. We provide a dataset generator that leverages the LDBC SNB prior generator to produce labelled social network graphs. We generate the datasets at different scale factors and make them and the benchmark available online[1]. Our main contribution is proposing nine queries as the core for a leaner and more focused subgraph query benchmark. In building the benchmark, we focus on five desiderata:

[1] https://github.com/ldbc/lsqb

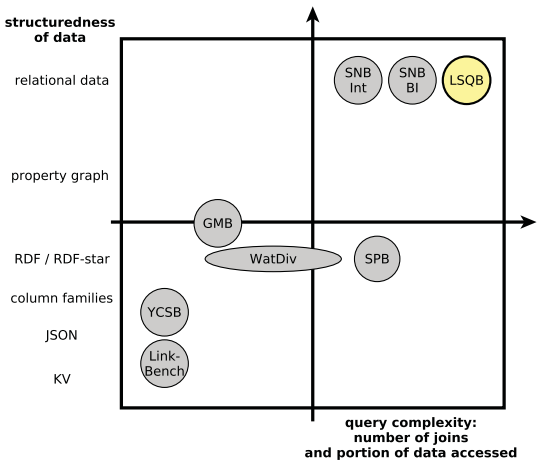


Figure 1: Landscape of DB benchmarks according to query complexity and data structuredness. The benchmark proposed in this paper, LSQB, is highlighted.

- *Simple*: contains a set of approachable queries that researchers can understand and implement.
- *Focused:* does not test the entire DBMS functionality and instead focuses on specific choke-points.
- *Scalable:* enables performance testing at the scale of billions of edges.
- *Portable:* provides a specification that is easy to map to the data models supported by different systems.
- *Representative:* contains a set of common use-cases found in real-world applications as opposed to contrived ones.

1.5 Paper Outline

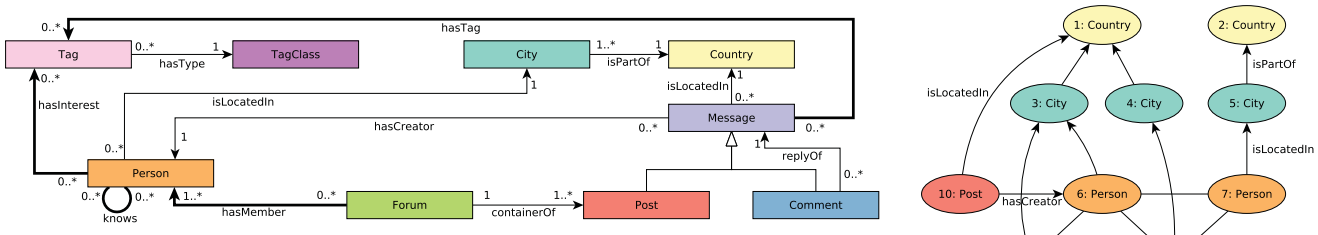
The rest of the paper is outlined as follows. In Section 2, we go over the design goals and provide the specification of the benchmark. In Section 3, we present preliminary performance numbers comparing two RDBMSs, and we give a sense of difficulty of the proposed queries. In Section 4, we review prior related work. Finally, in Section 5, we provide the gists of our findings, present possible extensions, along with our plan for future work.

2 BENCHMARK SPECIFICATION

In this section, we first describe the benchmark design and requirements allowing us to meet the desiderata mentioned in Section 1.4. Second, we discuss the workflow to execute the benchmark. Finally, we present the datasets and the queries.

2.1 Benchmark Design and Requirements

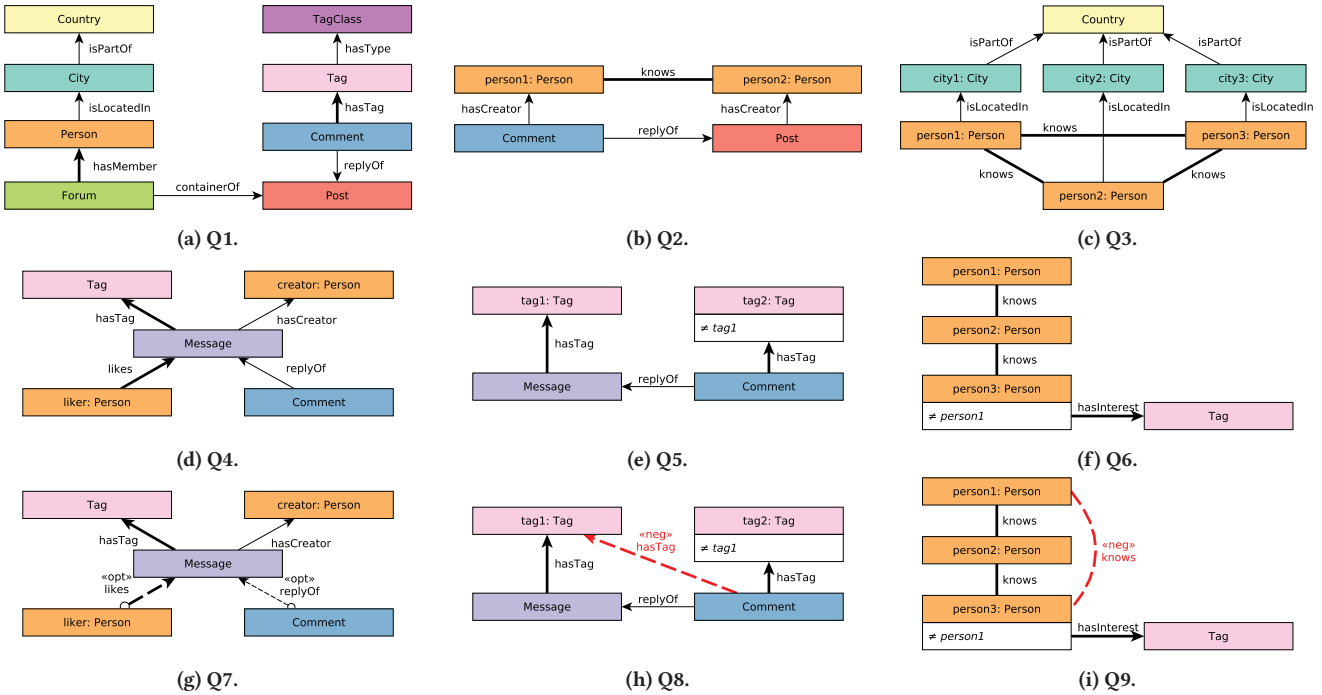
We start off with the need for the benchmark to be *simple* and *focused*. Subgraph queries contain a labelled structural graph pattern to match and contain possibly further relational operations, such as filtering, ordering, and aggregations. Matching a labelled structural graph pattern is referred to as subgraph matching and is a core part of subgraph queries. We make multi-way equi-joins, joining



(a) Graph schema visualized using a UML-style notation. Edge labels with many-to-many cardinality are depicted with thick lines. The knows edge labels represent undirected edges, while the rest of the edges are directed.

(b) Example graph.

Figure 2: Graph schema and example graph.



(a) Q1. (b) Q2. (c) Q3.

(d) Q4. (e) Q5. (f) Q6.

(g) Q7. (h) Q8. (i) Q9.

Figure 3: Visualization of the queries. Regular edges (joins) are denoted with solid black lines. Negative edges (antijoins) are denoted with dashed red lines and the «neg» keyword. Optional edges (left outer joins) are denoted with dashed black lines, the «opt» keyword, and the circle symbol ○ at the optional end of the edge. Thicker lines denote edge labels with many-to-many cardinality.

vertex records with their neighbors, the focus of LSQB, given that these operators are the most prominent in subgraph queries. This allows the benchmark to be focused on one single choke-point [9] and further allows us to limit the number of queries. In order for the benchmark to challenge systems in terms of scalability, we require at least one input data graph to be at the scale of at least one billion edges. This is in-line with current findings on the size of graphs analyzed in real workloads [47]. The queries are meant to be portable and therefore we specify them using graph patterns similar to LDBC SNB. Finally, the queries need to be representative and hence are designed based on real-world use cases on the dataset of choice.

2.2 Workflow

The benchmark consist of the following workflow: (i) the dataset is loaded to the system; (ii) the subgraph queries are executed sequentially; and (iii) two metrics are collected:

(1) The number of matches, i.e., the number of output tuples to validate the correctness of the implementation.
(2) The query runtime (including parsing, compiling, and returning results) for performance comparisons.

2.3 Datasets

2.3.1 Generator. We reuse the data generator of LDBC Datagen [4] with modifications. LDBC Datagen is a scalable graph generator

	SF3	SF10	SF30	SF100
number of vertices	11.3M	35.4M	103.1M	326.0M
number of edges	66.2M	217.0M	650.5M	2.1B
compressed size	0.3GB	1GB	1.8GB	5.8GB

Table 1: Dataset sizes.

which can produce graphs in increasing scale factors (SFs) up to 2.7B vertices and 17B edges. The LDBC Datagen produces a social network with a degree distribution of `Person` vertices similar to that found in Facebook, and introduces correlations between attributes [41], e.g., people are more likely to travel to neighbouring countries and post messages there.

2.3.2 Schema. Contemporary GDBMs support the property graph data model [45], which allows the use of properties (attributes) and labels (types) on both the vertices and edges of the graph. In the context of subgraph queries, properties are not a core focus, in fact, the experiments in the literature on subgraph query algorithms consistently omit using property-level data, which we explain in Section 4. Therefore, our generator omits all property-level information from the LDBC social graph except the vertex IDs. Our simplified schema (shown in Figure 2a) consists of 9 vertex types and 11 edge types. Figure 2b shows an example graph instance.

2.3.3 Datasets. We generated LDBC SNB datasets for scale factors 3, 10, 30, and 100. The key characteristics of the datasets are shown in Table 1, while detailed degree distributions are visualized in Appendix A. As the datasets lack property values, they are compact, with the size of SF100 being only 5.8GB in a compressed format.

2.3.4 Preprocessing Datasets. The datasets are provided as CSV files. Implementations are allowed to preprocess the datasets to fit their assumptions, e.g., they can assign new unique identifiers to vertices. Additionally, they can change the edge labels to ensure they are unique, for instance, the `isLocatedIn` edges between vertices of different labels can be distinguished as `Person_isLocatedIn_City` and `Comment_isLocatedIn_Country`.

2.4 Queries

2.4.1 Graph Patterns. We have defined 9 subgraph queries based on the graph patterns occurring in the LDBC SNB BI workload [4]. The queries are visualized in Figure 3. The first 6 queries look for matches of *basic graph patterns* [5], while the last 3 queries extend Q4, Q5, and Q6, respectively, to *complex graph patterns* with negative and optional edges. Subgraph queries can have different pattern matching semantics [5]. All of our queries are evaluated using *homomorphism*, i.e., repeated nodes and edges in the matching subgraph are allowed unless an explicit filter (such as `tag1` $\neq$ `tag2` in Q5) is set. If a particular system requires testing isomorphic pattern matching, adequate filters on edge IDs can be added. The SQL and openCypher [18] specification of the queries is given in Appendix B.

2.4.2 Query Operators. In all queries, the subgraph matching is followed by a `count(*)` aggregation, thus returning a single value with the total number of matches. This step is included to avoid serializing a large number of results which would stress the client protocol and not just the query engine which is the main focus of LSQB.

In terms of relational algebra, the first 6 queries can be formulated as SPJG (select-project-join-group by) queries. The last 3 queries require antijoin or outer join operations (rendering them SPOJG queries) which are often unsupported in early-stage systems. This decision was made to render the benchmark suitable for prototype systems by allowing them to focus on a contiguous block of queries (Q1–Q6). At the same time, the benchmark can examine the level of support for the more advanced operators (i.e., antijoins and outer joins) through the remaining queries (Q7–Q9).

3 PERFORMANCE EXPERIMENTS

We have implemented LSQB on multiple systems to study the benchmark's portability. In this section, we present our results on the two best performing systems. Along with the reported RDBMSs, we have implemented LSQB for 7+ different DBMSs. Unfortunately, our experiments showed significant performance limitations for all of these systems which we do not report.

3.1 Benchmark Setup

3.1.1 Systems. We report results of the benchmark on the two RDBMSs that managed to complete all or most of the workload at all scales: (i) HyPer [26] *(version 2019.2.6416)*, an HTAP RDBMS with a compiled runtime; and (ii) Umbra [34] *(version b273a006)*, a research prototype HTAP RDBMS with a compiled runtime that supports WCOJs [19].

3.1.2 Datasets. We generated four datasets, SF3, SF10, SF30, and SF100 using the LDBC Datagen and serialized them using the "raw" CSV layout. Next, we removed all attributes from the files and created one CSV file for each ⟨source vertex label, edge label, target vertex label⟩ triple (e.g., `Post_isLocatedIn_Country`).

3.1.3 Preprocessing and Loading. The RDBMSs loaded the data with the SQL `COPY` command. No indexes, integrity constraints, or uniqueness constrains were defined.

3.1.4 Environment. We executed the benchmark on a cloud virtual machine with 48 vCPU cores of an Intel Xeon Platinum 8272CL CPU, 384 GiB memory, 1.8 TB NVMe SSD disk (ext4 file system), Ubuntu 20.04 operating system with Linux kernel 5.4.0, and Docker 19.03. Both HyPer and Umbra ran in Docker containers using their default configuration and the queries were implemented in SQL.

3.1.5 Experiments. We ran HyPer and Umbra using all 48 available vCPU cores. Each query execution had a timeout of 5 minutes. The queries were executed in random order, with each query running 5 times for each scale factor.

3.2 Benchmark Results and Analysis

We executed the experiments and have cross-validated the results derived with HyPer, Umbra, and GDBMS systems. We visualized the query execution times in Figure 4. The figure contains a plot with a horizontal axis showing the scale factors, while the vertical axis shows the median query execution time (over 5 runs) using a logarithmic scale.

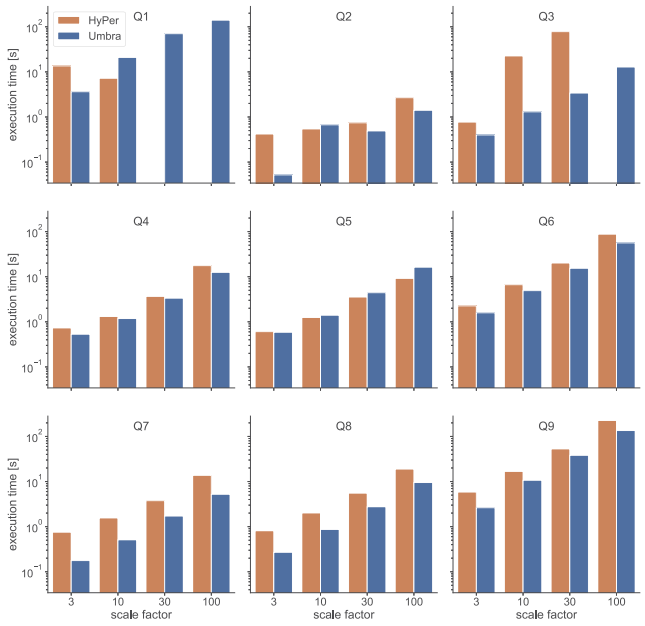


Figure 4: Query execution times on scale factors 3, 10, 30, and 100 for HyPer and Umbra, the two best performing sytems under benchmark.

3.2.1 Query Plans. We have investigated the query plans produced by HyPer and Umbra. We found that on the SF100, Umbra uses a *multiway join operator* [19] for all equi-join queries except Q2 where it uses binary joins for most scale factors. The multiway join was used in the last step of the query plan before the group-by aggregation. For the rest of the queries, Umbra used binary hash joins and right outer joins. HyPer used binary joins for all its queries.

3.2.2 Experimental Analysis. The query runtimes for scale factors 3, 10, 30, and 100 are shown in Figure 4. The results show that Umbra is able to complete all queries on all scale factors, while HyPer exhibits timeouts for Q1 and Q3.

Q1 and Q3 demonstrate the main difference of the execution engines of HyPer and Umbra: the latter supports multi-way WCOJs [19], while the former only supports binary joins. For Q1, Umbra consistently uses a multi-way join query plan, while HyPer uses different binary join plans for different SFs. The results show that the latter can result in suboptimal plans for this query: HyPer's evaluation of Q1 on SF3 is slower than SF10, and it times out for SF30 and SF100. Q3 has a cyclic subgraph, which again showcases the benefits of WCOJs. Umbra is already an order of magnitude faster than HyPer on SF30 and can complete the execution on SF100 while HyPer times out.

For the rest of the queries, the differences are less significant between the two systems. In particular, for queries Q4, Q5, and Q6, the performance gap between the two systems is less than 2× on large SFs. For Q2, Q7, Q8, and Q9, the differences are within an order of magnitude. For Q2, Umbra shows non-monotonic execution times with SF10 taking more time than SF30. We have found that for SF10, Umbra's optimizer switches to use a multi-way join, which is more costly than using binary joins (even though Q2 captures a cyclic pattern, it can be evaluated with binary joins optimally as it only has a single many-to-many relation).

For both Umbra and HyPer, the most challenging queries are Q6 and Q9 as these capture long cyclic patterns (two `knows` edges and one `hasTag` edge). To support such queries efficiently, database management systems would need to employ different techniques such as list-based processing, a limited form of factorization [38], which are currently not supported in the benchmarked systems. Note that we do not benchmark GraphflowDB's list-based processing engine [20] because it does not contain an optimizer. The results show that Q9, which uses an extra antijoin operator, is about $2-4\times$ more expensive to compute than Q6.

3.2.3 Findings. We summarize the findings of our implementations and performance experiments with the LSQB. We found that the benchmark can be executed in a reasonable amount of time: a complete execution of the multi-threaded benchmark (loading the data running each query 5 times) for scale factors 3, 10, 30, and 100, took less than 40 minutes with Umbra. This makes the benchmark suitable for quick iterations when optimizing systems.

The results showed that Umbra has a slight advantage for the majority of the queries. On Q3, the difference between the systems highlighted that to perform well across the queries of this subgraph query benchmark, systems need to have sufficiently sophisticated query optimizers and optimize for cyclic queries (e.g., by supporting WCOJs). Additionally, the structure of Q6 and Q9 raises the opportunity for employing factorized subgraph query processing for improving the performance of RDBMSs in these cases.

4 RELATED WORK

Table 2 summarizes the key experiment suites used for subgraph queries. We discuss influential benchmarks and experimental evaluation found in papers presenting subgraph isomorphism algorithms.

4.1 Benchmarks for DBMSs

Subgraph queries are frequently included in benchmarks targeting graph processing systems as well as programming contests. In this section, we summarize benchmarks where subgraph queries play a key role. For a survey of graph processing benchmarks, we point the reader to reference [10].

The *LDBC Social Network Benchmark* [4] defines two workloads. The *Interactive workload* [16] defines 14 complex read and 7 short read queries, all using subgraph matching. However, all queries are seeded, i.e., start from a given `Person` node, a pair of `Person` nodes, or a `Message` node, making their complexity limited. This is different from LSQB which relies on global, i.e., unseeded queries. The *Business Intelligence (BI)* [51] workload consists of 20 complex read queries, focusing on pattern matching with aggregation. BI queries include both cyclic and long acyclic graph queries, and while its queries are seeded, they touch a large portion of the graph. LSQB in comparison takes the position of being simpler and more focused and therefore only contains join operations with no aggregations other than `count(*)`. Recall that `count(*)` is added to avoid stressing client code and keeping the benchmark's stress only on the query processing engine.

name	vertex labels	edge labels	directed edges	properties	antijoins	outer joins	cyclic subgraphs	complex paths	aggregation	global queries	#queries	comment
LDBC SNB Interactive [16]	⊗	⊗	⊗	⊗	⊗	⊗	⊘	⊗	⊗	○	14 + 7	point queries starting in 1–2 vertices
LDBC SNB BI [4, 51]	⊗	⊗	⊗	⊗	⊗	⊗	⊘	⊗	⊗	⊘	20	queries touch on a large portion of the graph
SIGMOD'14 Programming Contest [15]	⊗	⊗	⊗	⊗	○	○	⊘	⊗	⊗	○	4	analytics on induced subgraphs
TTC'14 Movie Database case [23]	⊗	⊗	⊗	⊗	○	○	⊗	○	⊘	⊗	6	queries are part of a graph transformation
gMark [7]	⊗	⊗	⊗	⊗	○	○	⊘	⊗	⊘	○	⊛	has a configurable graph & query generator
Train Benchmark [52]	⊗	⊗	⊗	⊗	⊗	○	⊘	○	○	⊗	6	query results sets are small
GMB (Graph Micro Benchmark) [31]	⊗	⊗	⊗	⊗	○	○	⊘	⊗	○	⊘	22	focuses on transactional workloads
DBMSs for pattern matching study [42]	⊗	⊗	○	○	○	○	⊗	○	○	⊗	14 × 20	
subgraph isomorphism survey [29]	⊗	⊗	○	○	○	○	⊘	○	○	⊗	6 × 1000	
subgraph matching survey [49]	⊗	○	○	○	○	○	⊗	○	○	⊗	12 × 9 × 200	
Core-Forest-Leaf decomposition [8]	⊗	○	⊗	○	○	○	⊗	○	○	⊗	8 × 100	
RapidMatch [50]	⊗	○	○	○	○	○	⊗	○	⊘	⊗	7 + 2 × 200	small and large query sets
worst-case optimal joins [36]	○	⊗	○	○	○	○	⊗	○	○	⊗	–	theoretical paper
graph pattern matching with joins [37]	○	○	○	○	○	○	⊗	○	⊘	⊗	10	
EmptyHeaded [1]	○	○	○	○	○	○	⊗	○	○	⊘	3 + 14	3 subgraph queries + LUBM
mix of multi-way and binary joins [32]	○	⊗	⊗	○	○	○	⊗	○	⊗	⊗	14	uses random labels in its experiments
Labelled Subgraph Query Benchmark (LSQB)	⊗	⊗	⊗	○	⊘	⊘	⊘	○	⊘	⊗	9	

Table 2: Experiments defined by papers presenting (1) graph benchmark specifications, (2) subgraph matching algorithms, and (3) worst-cast optimal join algorithms. For benchmarks, the columns *vertex labels, edge labels, directed edges, and attributes* denote whether the feature is used in the queries of the benchmark. For the rest of the entries, they denote whether the proposed algorithm is designed to handle a certain data model feature. Notation – ⊗: yes, ○: no, ⊘: to some extent (*cyclic subgraphs:* the benchmark defines only a few/simple cyclic queries, *aggregation*: only `count(*)` is used, *global queries*: some queries touch a large portion of the graph); ⊛: the benchmark provides a query generator.

The *Train Benchmark* (TB) [52] captures a validation scenario for imposed constraints, often complex, during the development of safety-critical systems. The data and constraints follow the property graph model where the validation process matches absent graph patterns which captures constraint violations. TB executes global subgraph queries with a few results but potentially large intermediate results. LSQB and TB differ primarily in their use cases and targeted queries. LSQB processes one-time queries for social network analysis while TB processes continuous queries.

Many transactional graph processing benchmarks exist, e.g., the *Graph Micro Benchmark* [31] defines a comprehensive micro-benchmark of 35 operations. It focuses on transactional GDBMSs, that is, it includes also inserts, updates, and deletes. Moreover, instead of considering queries with a complex structure, it opts for a broader set of primitive operators. LSQB on the other hand focuses specifically on read-intensive subgraph matching and contains no updates.

Other than the benchmarks mentioned above, experimental studies have also defined their own set of queries. For instance, a study on Graph Pattern Matching [42] benchmarked subgraph queries on four DMBSs, including both relational and graph systems. Their largest graph contained 10K vertices and 0.5M edges, while their most complex schema used 5 vertex labels and 5 edge labels. They concluded that (as of 2014) none of the benchmarked systems were suitable for graph pattern matching workloads. With exception of this last study, no previous paper explicitly focused on testing DBMSs for the task of graph pattern matching. Moreover, our work is the first to propose a benchmark for this type of workload at the scale of billion edges and containing more vertex and edge labels.

4.2 Subgraph Matching Experiments

Due to the lack of simple benchmarks that focus on graph pattern matching, experimental evaluation in the literature for techniques targeting subgraph query processing contain a fragmented panorama of benchmarks. Such benchmarks follow one of two types of workloads that are prominent in the literature as observed in prior work [50]: (i) *full enumeration workloads* on datasets with millions or even billions of edges and queries with few query vertices, and requiring exhaustive tuple enumeration, such as fraud detection; and (ii) *exploration-based workloads* on datasets with few thousand edges and tens of query vertices. LSQB focuses on the former category, i.e., on *full enumeration subgraph query workloads*.

Jinsoo Lee et al. have reimplemented and compared five state-of-the-art subgraph matching algorithms [29]. They used undirected graphs with vertex and edge labels, and ran experiments on 6 datasets with 1000 randomly generated queries for each. A more recent similar study [49] compares a mix of old and recent subgraph matching algorithms for a total of seven algorithms and proposes proposes an algorithm that combines the best approaches

of each of the studied algorithms. In both studies, as well as in many proposed subgraph query algorithms evaluations, e.g., CFL [8], the workload is an exploration-based workload. A notable exception is RapidMatch [50], which uses two sets of queries: 2×200 large queries (for graph exploration) and 7 small queries (similar in size to LSQB queries). RapidMatch's evaluation however does not use directed edges and contains uniformly generated labels for 5 out of 7 graphs used.

A number of implementations have recently adopted worst-case optimal join (*WCOJs*) algorithms [36] for subgraph matching [1, 37]. In these papers, the experiments employ rather simple graphs. The proposed dynamic programming optimizer for subgraph queries focusing on mixing binary and multi-way join operators in reference [32], is evaluated using randomly labelled data and query graphs.

4.3 Join Ordering

In an RDBMS, LSQB's main choke-point is translated into a number of joins. Therefore, a crucial challenge is that of picking a good join order, referred to as a query vertex order or a query edge order in graph terms. The *Join Order Benchmark* (JOB) [30], which defines a set of 113 analytical SPJ queries over an IMDB dataset, each containing 3 to 16 joins per query, provides a similar challenge for RDBMSs. On JOB, cardinality misestimation of 3+ orders of magnitude were routinely observed in all systems that were benchmarked. Since JOB was introduced, a wide variety of techniques have been proposed to address the problem of join ordering [22, 40]. LSQB also poses a hard cardinality estimation problem since it contains many joins and skewed distributions.

5 SUMMARY

Conclusion. By examining the existing literature and benchmarks, we found that there is a significant gap between the experiments currently used on subgraph queries and the requirements posed by graph processing benchmarks. Our benchmark, LSQB, intends to provide a simple and portable benchmark suite that can be used by both academic researchers and industry database engineers. LSQB has already seen some adoption and is used by 5+ triplestore/GDBMS vendors for internal performance tests.

Potential Extensions. While we designed the benchmark to be simple, we expect that users targeting more mature systems will want to extend it to include common DBMS features such as update operations. Building on the LDBC Datagen allows users to make such extensions with a reasonable development effort, e.g., updates can be supported by using the *temporal attributes* produced by Datagen [54].

Future Work. As a continuation of this work, we plan to design a complementary micro-benchmark that focuses on *path queries*. Defining path queries with predictable runtimes raises unique challenges, as these queries are sensitive to the selected start vertices. The focus would be in particular on long acyclic patterns, unweighted/weighted shortest paths, and regular path queries.

ACKNOWLEDGMENTS

We would like to thank Michael Freitag for providing us assistance in using Umbra. Matteo Lissandrini is supported by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 838216. Gábor Szárnyas is supported by the SQIREL-GRAPHS NWO project.

REFERENCES

[1] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4 (2017), 20:1–20:44. https://doi.org/10.1145/3129246

[2] Günes Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. 2014. Diversified Stress Testing of RDF Data Management Systems. In *ISWC*. Springer, 197–212. https://doi.org/10.1007/978-3-319-11964-9_13

[3] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed Evaluation of Subgraph Queries Using Worst-case Optimal and Low-Memory Dataflows. *Proc. VLDB Endow.* 11, 6 (2018), 691–704. https://doi.org/10.14778/3184470.3184473

[4] Renzo Angles et al. 2020. The LDBC Social Network Benchmark. *CoRR* abs/2001.02299 (2020). http://arxiv.org/abs/2001.02299

[5] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5 (2017), 68:1–68:40. https://doi.org/10.1145/3104031

[6] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: A database benchmark based on the Facebook social graph. In *SIGMOD*. 1185–1196. https://doi.org/10.1145/2463676.2465296

[7] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurélien Lemay, and Nicky Advokaat. 2017. gMark: Schema-Driven Generation of Graphs and Queries. *IEEE Trans. Knowl. Data Eng.* 29, 4 (2017), 856–869. https://doi.org/10.1109/TKDE.2016.2633993

[8] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products. In *SIGMOD*. ACM, 1199–1214. https://doi.org/10.1145/2882903.2915236

[9] Peter A. Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *TPCTC*. Springer, 61–76. https://doi.org/10.1007/978-3-319-04936-6_5

[10] Angela Bonifati, George H. L. Fletcher, Jan Hidders, and Alexandru Iosup. 2018. A Survey of Benchmarks for Graph-Processing Systems. In *Graph Data Management, Fundamental Issues and Recent Developments*. Springer, 163–186. https://doi.org/10.1007/978-3-319-96193-4_6

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*. ACM, 143–154. https://doi.org/10.1145/1807128.1807152

[12] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2019. TigerGraph: A Native MPP Graph Database. *CoRR* abs/1901.08248 (2019). http://arxiv.org/abs/1901.08248

[13] Songyun Duan, Anastasios Kementsietsidis, Kavitha Srinivas, and Octavian Udrea. 2011. Apples and oranges: A comparison of RDF benchmarks and real RDF datasets. In *SIGMOD*. ACM, 145–156. https://doi.org/10.1145/1989323.1989340

[14] Márton Elekes, János Benjamin Antal, and Gábor Szárnyas. 2020. An analysis of the SIGMOD 2014 Programming Contest: Complex queries on the LDBC social network graph. *CoRR* abs/2010.12243 (2020). arXiv:2010.12243 https://arxiv.org/abs/2010.12243

[15] Márton Elekes, János Benjamin Antal, and Gábor Szárnyas. 2020. An analysis of the SIGMOD 2014 Programming Contest: Complex queries on the LDBC social network graph. *CoRR* abs/2010.12243 (2020). https://arxiv.org/abs/2010.12243

[16] Orri Erling, Alex Averbuch, Josep-Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD*. ACM, 619–630. https://doi.org/10.1145/2723372.2742786

[17] Santo Fortunato. 2009. Community detection in graphs. *CoRR* abs/0906.0612 (2009). http://arxiv.org/abs/0906.0612

[18] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *SIGMOD*. ACM, 1433–1445. https://doi.org/10.1145/3183713.3190657

[19] Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting Worst-Case Optimal Joins in Relational Database Systems. *Proc. VLDB Endow.* 13, 11 (2020), 1891–1904. http://www.vldb.org/pvldb/vol13/p1891-freitag.pdf

[20] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. 2021. Integrating Column-Oriented Storage and Query Processing Techniques Into Graph Database Management Systems. *CoRR* abs/2103.02284 (2021). https://arxiv.org/abs/2103.02284
[21] Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabiuk, Quannan Li, and Jimmy J. Lin. 2014. Real-Time Twitter Recommendation: Online Motif Detection in Large Dynamic Graphs. *Proc. VLDB Endow.* 7, 13 (2014), 1379–1380. https://doi.org/10.14778/2733004.2733010
[22] Axel Hertzschuch, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2021. Simplicity Done Right for Join Ordering. In *CIDR*. http://cidrdb.org/cidr2021/papers/cidr2021_paper01.pdf
[23] Tassilo Horn, Christian Krause, and Matthias Tichy. 2014. The TTC 2014 Movie Database Case. In *Transformation Tool Contest at STAF*, Vol. 1305. 93–97. http://ceur-ws.org/Vol-1305/paper2.pdf
[24] Alexandru Iosup et al. 2020. The LDBC Graphalytics Benchmark. *CoRR* abs/2011.15028 (2020). https://arxiv.org/abs/2011.15028
[25] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *SIGMOD*. ACM, 1695–1698. https://doi.org/10.1145/3035918.3056445
[26] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*. IEEE Computer Society, 195–206. https://doi.org/10.1109/ICDE.2011.5767867
[27] Venelin Kotsev, Nikos Minadakis, Vassilis Papakonstantinou, Orri Erling, Irini Fundulaki, and Atanas Kiryakov. 2016. Benchmarking RDF Query Engines: The LDBC Semantic Publishing Benchmark. In *BLINK at ISWC*. http://ceur-ws.org/Vol-1700/paper-01.pdf
[28] Longbin Lai et al. 2019. Distributed Subgraph Matching on Timely Dataflow. *Proc. VLDB Endow.* 12, 10 (2019), 1099–1112. https://doi.org/10.14778/3339490.3339494
[29] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. 2012. An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases. *Proc. VLDB Endow.* 6, 2 (2012), 133–144. https://doi.org/10.14778/2535568.2448946
[30] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 204–215. https://doi.org/10.14778/2850583.2850594
[31] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. 2018. Beyond Macrobenchmarks: Microbenchmark-based Graph Database Evaluation. *Proc. VLDB Endow.* 12, 4 (2018), 390–403. https://doi.org/10.14778/3297753.3297759
[32] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proc. VLDB Endow.* 12, 11 (2019), 1692–1704. https://doi.org/10.14778/3342263.3342643
[33] Neo4j. 2021. Property Graph Model. https://neo4j.com/developer/graph-database.
[34] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf
[35] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case Optimal Join Algorithms. *J. ACM* 65, 3 (2018), 16:1–16:40. https://doi.org/10.1145/3180143
[36] Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2013. Skew strikes back: New developments in the theory of join algorithms. *SIGMOD Rec.* 42, 4 (2013), 5–16. https://doi.org/10.1145/2590989.2590991
[37] Dung T. Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2015. Join Processing for Graph Patterns: An Old Dog with New Tricks. In *GRADES at SIGMOD*. ACM, 2:1–2:8. https://doi.org/10.1145/2764947.2764948
[38] Dan Olteanu and Maximilian Schleich. 2016. Factorized Databases. *SIGMOD Rec.* 45, 2 (2016), 5–16. https://doi.org/10.1145/3003665.3003667
[39] M. Tamer Özsu. 2019. Graph Processing: A Panoramic View and Some Open Problems. https://vldb2019.github.io/files/VLDB19-keynote-1-slides.pdf.
[40] Matthew Perron, Zeyuan Shang, Tim Kraska, and Michael Stonebraker. 2019. How I Learned to Stop Worrying and Love Re-optimization. In *ICDE*. IEEE, 1758–1761. https://doi.org/10.1109/ICDE.2019.00191
[41] Minh-Duc Pham, Peter A. Boncz, and Orri Erling. 2012. S3G2: A Scalable Structure-Correlated Social Graph Generator. In *TPCTC*. Springer, 156–172. https://doi.org/10.1007/978-3-642-36727-4_11
[42] Nataliia Pobiedina, Stefan Rümmele, Sebastian Skritek, and Hannes Werthner. 2014. Benchmarking Database Systems for Graph Pattern Matching. In *Database and Expert Systems Applications (DEXA)*. Springer, 226–241. https://doi.org/10.1007/978-3-319-10073-9_18
[43] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time Constrained Cycle Detection in Large Dynamic Graphs. *Proc. VLDB Endow.* 11, 12 (2018), 1876–1888. https://doi.org/10.14778/3229863.3229874
[44] Raghavan Raman, Oskar van Rest, Sungpack Hong, Zhe Wu, Hassan Chafi, and Jay Banerjee. 2014. PGX.ISO: Parallel and Efficient In-Memory Engine for Subgraph Isomorphism. In *GRADES at SIGMOD*. ACM, 5:1–5:6. https://doi.org/10.1145/2621934.2621939
[45] Marko A. Rodriguez and Peter Neubauer. 2010. Constructions from dots and lines. *Bulletin of the American Society for Information Science and Technology* 36, 6 (2010), 35–41. https://doi.org/10.1002/bult.2010.1720360610
[46] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. 2013. The Graph Story of the SAP HANA Database. In *BTW*. GI, 403–420. https://dl.gi.de/20.500.12116/17334
[47] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: Extended survey. *VLDB J.* 29, 2 (2020), 595–618. https://doi.org/10.1007/s00778-019-00548-x
[48] Sherif Sakr et al. 2020. The Future is Big Graphs! A Community View on Graph Processing Systems. *CoRR* abs/2012.06171 (2020). https://arxiv.org/abs/2012.06171
[49] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An In-depth Study. In *SIGMOD*. ACM, 1083–1098. https://doi.org/10.1145/3318464.3380581
[50] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. RapidMatch: A Holistic Approach to Subgraph Query Processing. *VLDB* (2020). https://doi.org/10.14778/3425879.3425888
[51] Gábor Szárnyas et al. 2018. An early look at the LDBC Social Network Benchmark's Business Intelligence workload. In *GRADES-NDA at SIGMOD/PODS*. ACM, 9:1–9:11. https://doi.org/10.1145/3210259.3210268
[52] Gábor Szárnyas, Benedek Izsó, István Ráth, and Dániel Varró. 2018. The Train Benchmark: Cross-technology performance evaluation of continuous model queries. *Softw. Syst. Model.* 17, 4 (2018), 1365–1393. https://doi.org/10.1007/s10270-016-0571-8
[53] Yuanyuan Tian, En Liang Xu, Wei Zhao, Mir Hamid Pirahesh, Suijun Tong, Wen Sun, Thomas Kolanko, Md. Shahidul Haque Apu, and Huijuan Peng. 2020. IBM Db2 Graph: Supporting Synergistic and Retrofittable Graph Queries Inside IBM Db2. In *SIGMOD*. ACM, 345–359. https://doi.org/10.1145/3318464.3386138
[54] Jack Waudby, Benjamin A. Steer, Arnau Prat-Pérez, and Gábor Szárnyas. 2020. Supporting Dynamic Graphs and Temporal Entity Deletions in the LDBC Social Network Benchmark's Data Generator. In *GRADES-NDA at SIGMOD*. ACM, 8:1–8:8. https://doi.org/10.1145/3398682.3399165
[55] Jim Webber. 2012. A programmatic introduction to Neo4j. In *SPLASH*. ACM, 217–218. https://doi.org/10.1145/2384716.2384777
[56] Sungheun Wi, Wook-Shin Han, Chu-Ho Chang, and Kihong Kim. 2020. Towards Multi-way Join Aware Optimizer in SAP HANA. *Proc. VLDB Endow.* 13, 12 (2020), 3019–3031. http://www.vldb.org/pvldb/vol13/p3019-wi.pdf

A DEGREE DISTRIBUTIONS

The degree distributions of the SF10 graph are shown in Figure 5.

B QUERY SPECIFICATIONS

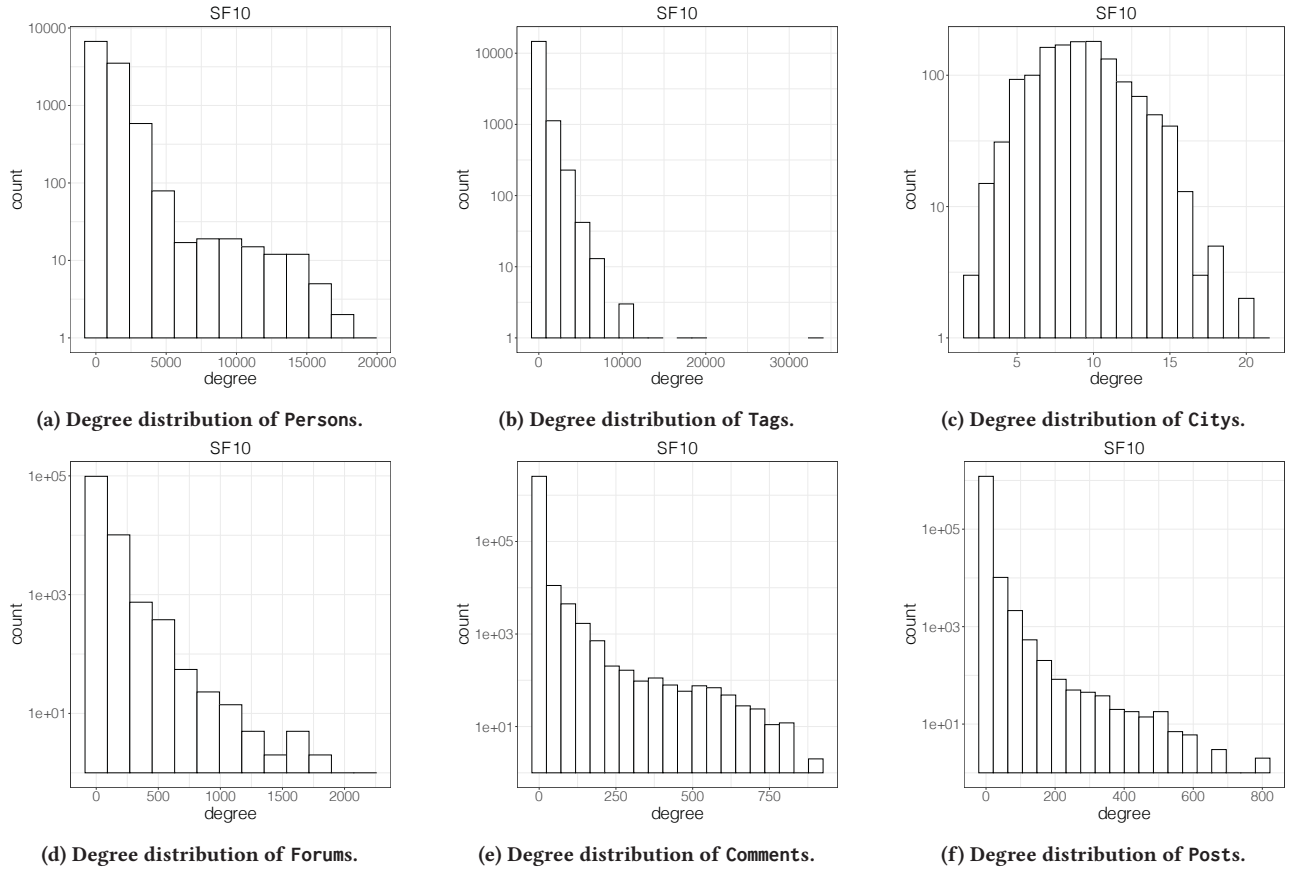
We present the specification of the queries in Cypher [18] and in SQL.

```
MATCH (:Country)<-[:IS_PART_OF]-(:City)<-[:IS_LOCATED_IN]-(:
      Person)<-[:HAS_MEMBER]-(:Forum)-[:CONTAINER_OF]->(:Post
      )<-[:REPLY_OF]-(:Comment)-[:HAS_TAG]->(:Tag)-[:HAS_TYPE
      ]->(:TagClass)
RETURN count(*) AS count
```

Cypher implementation of Q1.

```
MATCH
  (person1:Person)-[:KNOWS]-(person2:Person),
  (person1)<-[:HAS_CREATOR]-(comment:Comment)-[:REPLY_OF]->(
     post:Post)-[:HAS_CREATOR]->(person2)
RETURN count(*) AS count
```

Cypher implementation of Q2.



(a) Degree distribution of `Persons`.

(b) Degree distribution of `Tags`.

(c) Degree distribution of `Citys`.

(d) Degree distribution of `Forums`.

(e) Degree distribution of `Comments`.

(f) Degree distribution of `Posts`.

Figure 5: Degree distributions for vertices with a given label.

```
MATCH (country:Country)
MATCH (person1:Person)-[:IS_LOCATED_IN]->(city1:City)-[:
    IS_PART_OF]->(country)
MATCH (person2:Person)-[:IS_LOCATED_IN]->(city2:City)-[:
    IS_PART_OF]->(country)
MATCH (person3:Person)-[:IS_LOCATED_IN]->(city3:City)-[:
    IS_PART_OF]->(country)
MATCH (person1)-[:KNOWS]-(person2)-[:KNOWS]-(person3)-[:
    KNOWS]-(person1)
RETURN count(*) AS count
```

Cypher implementation of Q3.

```
MATCH (:Tag)<-[:HAS_TAG]-(message:Message)-[:HAS_CREATOR]-(
    creator:Person),
  (message)<-[:LIKES]-(liker:Person),
  (message)<-[:REPLY_OF]-(comment:Comment)
RETURN count(*) AS count
```

Cypher implementation of Q4.

```
MATCH (tag1:Tag)<-[:HAS_TAG]-(message:Message)<-[:REPLY_OF
    ]-(comment:Comment)-[:HAS_TAG]->(tag2:Tag)
WHERE tag1 <> tag2
RETURN count(*) AS count
```

Cypher implementation of Q5.

```
MATCH (person1:Person)-[:KNOWS]-(person2:Person)-[:KNOWS]-(
    person3:Person)-[:HAS_INTEREST]->(tag:Tag)
WHERE person1 <> person3
RETURN count(*) AS count
```

Cypher implementation of Q6.

```
MATCH (:Tag)<-[:HAS_TAG]-(message:Message)-[:HAS_CREATOR]-(
    creator:Person)
OPTIONAL MATCH (message)<-[:LIKES]-(liker:Person)
OPTIONAL MATCH (message)<-[:REPLY_OF]-(comment:Comment)
RETURN count(*) AS count
```

Cypher implementation of Q7.

```
MATCH (tag1:Tag)<-[:HAS_TAG]-(message:Message)<-[:REPLY_OF
    ]-(comment:Comment)-[:HAS_TAG]->(tag2:Tag)
WHERE NOT (comment)-[:HAS_TAG]->(tag1)
  AND tag1 <> tag2
RETURN count(*) AS count
```

Cypher implementation of Q8.

```
MATCH (person1:Person)-[:KNOWS]-(person2:Person)-[:KNOWS]-(
     person3:Person)-[:HAS_INTEREST]->(tag:Tag)
WHERE NOT (person1)-[:KNOWS]-(person3)
  AND person1 <> person3
RETURN count(*) AS count
```

Cypher implementation of Q9.

```
SELECT count(*)
FROM Country
JOIN City ON City.isPartOf_Country = Country.id
JOIN Person ON Person.isLocatedIn_City = City.id
JOIN Forum_hasMember_Person ON Forum_hasMember_Person.
     hasMember_Person = Person.id
JOIN Forum ON Forum.id = Forum_hasMember_Person.id
JOIN Post ON Post.Forum_containerOf = Forum.id
JOIN Comment ON Comment.replyOf_Post = Post.id
JOIN Comment_hasTag_Tag ON Comment_hasTag_Tag.id = Comment.
     id
JOIN Tag ON Tag.id = Comment_hasTag_Tag.hasTag_Tag
JOIN TagClass ON Tag.hasType_TagClass = TagClass.id;
```

SQL implementation of Q1.

```
SELECT count(*)
FROM Person_knows_Person
JOIN Comment ON Person_knows_Person.Person1Id = Comment.
     hasCreator_Person
JOIN Post ON Person_knows_Person.Person2Id = Post.
     hasCreator_Person
 AND Comment.replyOf_Post = Post.id;
```

SQL implementation of Q2.

```
SELECT count(*)
FROM Country
JOIN City AS CityA
  ON CityA.isPartOf_Country = Country.id
JOIN City AS CityB
  ON CityB.isPartOf_Country = Country.id
JOIN City AS CityC
  ON CityC.isPartOf_Country = Country.id
JOIN Person AS PersonA
  ON PersonA.isLocatedIn_City = CityA.id
JOIN Person AS PersonB
  ON PersonB.isLocatedIn_City = CityB.id
JOIN Person AS PersonC
  ON PersonC.isLocatedIn_City = CityC.id
JOIN Person_knows_Person AS pkp1
  ON pkp1.Person1Id = personA.id
 AND pkp1.Person2Id = personB.id
JOIN Person_knows_Person AS pkp2
  ON pkp2.Person1Id = personB.id
 AND pkp2.Person2Id = personC.id
JOIN Person_knows_person AS pkp3
  ON pkp3.Person1Id = personC.id
 AND pkp3.Person2Id = personA.id;
```

SQL implementation of Q3.

```
SELECT count(*)
FROM Message_hasTag_Tag
JOIN Message_hasCreator_Person
  ON Message_hasTag_Tag.MessageId =
     Message_hasCreator_Person.MessageId
JOIN Comment_replyOf_Message
  ON Comment_replyOf_Message.ParentMessageId =
     Message_hasTag_Tag.MessageId
JOIN Person_likes_Message
  ON Person_likes_Message.MessageId = Message_hasTag_Tag.
     MessageId;
```

SQL implementation of Q4.

```
SELECT count(*)
FROM Message_hasTag_Tag
JOIN Comment_replyOf_Message
  ON Message_hasTag_Tag.MessageId = Comment_replyOf_Message.
     ParentMessageId
JOIN Comment_hasTag_Tag AS cht
  ON Comment_replyOf_Message.CommentId = cht.id
WHERE Message_hasTag_Tag.hasTag_Tag != cht.hasTag_Tag;
```

SQL implementation of Q5.

```
SELECT count(*)
FROM Person_knows_Person pkp1
JOIN Person_knows_Person pkp2
  ON pkp1.Person2Id = pkp2.Person1Id
 AND pkp1.Person1Id != pkp2.Person2Id
JOIN Person_hasInterest_Tag
  ON pkp2.Person2Id = Person_hasInterest_Tag.id;
```

SQL implementation of Q6.

```
SELECT count(*)
FROM Message_hasTag_Tag
JOIN Message_hasCreator_Person
  ON Message_hasTag_Tag.MessageId =
     Message_hasCreator_Person.MessageId
LEFT JOIN Comment_replyOf_Message
  ON Comment_replyOf_Message.ParentMessageId =
     Message_hasTag_Tag.MessageId
LEFT JOIN Person_likes_Message
  ON Person_likes_Message.MessageId = Message_hasTag_Tag.
     MessageId;
```

SQL implementation of Q7.

```
SELECT count(*)
FROM Message_hasTag_Tag
JOIN Comment_replyOf_Message
  ON Message_hasTag_Tag.MessageId = Comment_replyOf_Message.
     ParentMessageId
JOIN Comment_hasTag_Tag AS cht1
  ON Comment_replyOf_Message.CommentId = cht1.id
LEFT JOIN Comment_hasTag_Tag AS cht2
       ON Message_hasTag_Tag.hasTag_Tag = cht2.hasTag_Tag
      AND Comment_replyOf_Message.CommentId = cht2.id
    WHERE Message_hasTag_Tag.hasTag_Tag != cht1.hasTag_Tag
      AND cht2.hasTag_Tag IS NULL;
```

SQL implementation of Q8.

```
SELECT count(*)
FROM Person_knows_Person pkp1
JOIN Person_knows_Person pkp2
  ON pkp1.Person2Id = pkp2.Person1Id
 AND pkp1.Person1Id != pkp2.Person2Id
JOIN Person_hasInterest_Tag
  ON pkp2.Person2Id = Person_hasInterest_Tag.id
LEFT JOIN Person_knows_Person pkp3
       ON pkp3.Person1Id = pkp1.Person1Id
      AND pkp3.Person2Id = pkp2.Person2Id
    WHERE pkp3.Person1Id IS NULL;
```

SQL implementation of Q9.

C NUMBER OF MATCHES

query	SF3	SF10	SF30	SF100
Q1	817.1M	3.1B	10.5B	38.3B
Q2	3.3M	11.4M	34.9M	115.4M
Q3	3.2M	15.0M	60.7M	255.7M
Q4	50.0M	191.2M	654.5M	2.4B
Q5	41.6M	141.7M	432.7M	1.4B
Q6	6.3B	23.7B	79.9B	291.4B
Q7	85.2M	312.8M	1.0B	3.7B
Q8	20.8M	70.8M	216.2M	710.8M
Q9	6.0B	22.9B	76.9B	280.0B

Table 3: Number of matching subgraphs per query.

The number of matches for each query–scale factor pair is shown in Table 3.