The LDBC Social Network Benchmark: Business Intelligence Workload

Gábor Szárnyas
CWI
gabor.szarnyas@cwi.nl

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

Benjamin A. Steer
Pometry
ben.steer@pometry.com

Dávid Szakállas
Independent contributor
david.szakallas@gmail.com

Altan Birler
Technische Universität München
altan.birler@tum.de

Mingxi Wu
TigerGraph
mingxi.wu@tigergraph.com

Yuchen Zhang
TigerGraph
yuchen.zhang@tigergraph.com

Peter Boncz
CWI
boncz@cwi.nl

ABSTRACT

The Social Network Benchmark's Business Intelligence workload (SNB BI) is a comprehensive graph OLAP benchmark targeting analytical data systems capable of supporting graph workloads. This paper marks the finalization of almost a decade of research in academia and industry via the Linked Data Benchmark Council (LDBC). SNB BI advances the state-of-the art in synthetic and scalable analytical database benchmarks in many aspects. Its base is a sophisticated data generator, implemented on a scalable distributed infrastructure, that produces a social graph with small-world phenomena, whose value properties follow skewed and correlated distributions and where values correlate with structure. This is a temporal graph where all nodes and edges follow lifespan-based rules with temporal skew enabling realistic and consistent temporal inserts and (recursive) deletes. The query workload exploiting this skew and correlation is based on LDBC's "choke point"-driven design methodology and will entice technical and scientific improvements in future (graph) database systems. SNB BI includes the first adoption of "parameter curation" in an analytical benchmark, a technique that ensures stable runtimes of query variants across different parameter values. Two performance metrics characterize peak single-query performance (power) and sustained concurrent query throughput. To demonstrate the portability of the benchmark, we present experimental results on a relational and a graph DBMS. Note that these do not constitute an official LDBC Benchmark Result – only audited results can use this trademarked term.

PVLDB Reference Format:
Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter Boncz. The LDBC Social Network Benchmark: Business Intelligence Workload. PVLDB, 16(4): 877 - 890, 2022.
doi:10.14778/3574245.3574270

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ldbc/ldbc_snb_bi/releases/tag/v1.0.3.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 16, No. 4 ISSN 2150-8097.
doi:10.14778/3574245.3574270

Table 1: The SNB BI workload fills in the space between LDBC SNB Interactive and LDBC Graphalytics. It is a graph OLAP workload focusing on queries on a labelled attributed graph with temporal changes (inserts and deletes), targeting systems with domain-specific query languages. We denote the data models and features covered, and whether a language is *capable of implementing* and *allowed to implement* a given benchmark. *Notation:* ⊗: yes, ○: no, ⊘: limited coverage.

	OLTP	**OLAP**	**algorithms**
LDBC benchmark	SNB Interactive	SNB BI	Graphalytics
labelled attributed graph	⊗	⊗	○
insert operations	⊗	⊗	○
delete operations	○	⊗	○
challenging joins	○	⊗	⊘
cheapest path finding	○	⊗	⊗
inter-query parallelism	required	optional	not allowed
query footprint	small	large	all data
SQL with recursion	⊗	⊗	○
GQL, SQL/PGQ, Cypher	⊗	⊗	○
GSQL	⊗	⊗	⊗
SPARQL+path extension	⊗	⊗	○
imperative API	⊗	○	⊗

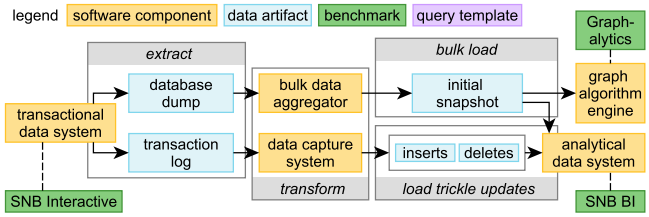
1 INTRODUCTION

Analyzing the connection patterns in graphs is a steadily expanding use case in data analytics and is projected to still grow considerably in importance [57]. It is reflected in the increasing role of graph-shaped data as represented in data models such as (initially) RDF and increasingly *property graphs* [5]. While graph analytics is often associated with obviously graph-intensive application domains that manage data representing social networks, telecommunication networks, and enterprise knowledge graphs [60], graph challenges are also found in traditional relational data warehouses and modern data lakes, where implicit graphs lurk in the connection patterns formed between tables that refer to each other through joins along relationships, esp. along many-to-many relationships. Practitioners, data system builders, and researchers are increasingly focusing on graph analysis questions [56], performing tasks such as fraud detection, recommendation, historical analysis, and root-cause analysis.

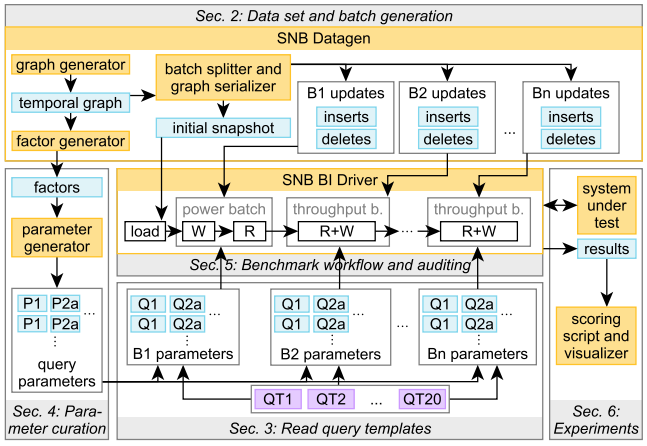
The Linked Data Benchmark Council. To expedite the evolution of the modern graph data management stack, a group of industry and academic organizations founded the Linked Data Benchmark Council (LDBC) in 2012, originally as a European Union-funded project.

Since 2015 LDBC has been an independent non-profit benchmarking organization that provides a set of standard benchmarks to make graph processing performance measurable and thus facilitate competition between vendors. These benchmarks were designed using a *choke point*-based methodology [11, 19], which ensures the coverage of challenging data management features. LDBC thus fulfills a role similar to the Transaction Processing Performance Council (TPC), which defined a number of influential benchmarks (e.g. TPC-C [67], TPC-H [68]) and greatly contributed to the rapid improvement of RDBMS performance [23].

Graph query languages. Initially, there were no standard data models and query languages for graph DBMSs, which hindered development of benchmarks. LDBC therefore expanded from benchmarking only to fomenting the development of standards, and established a liaison with the ISO SQL working group in 2016. Building on, among others, the LDBC G-CORE [7] proposal, this liaison helped ISO design the native Graph Query Language (GQL) as well as the SQL/PGQ (Property Graph Queries) extension, which will be part of the next SQL standard, expected in 2023 [16].



(a) Model system implementing an ETL process from a transactional to an analytical data system and the systems targeted by each LDBC benchmark.



(b) Main components of SNB BI, and their connection to the workflow executed by the benchmark driver on the system under test. Writes (W) apply update operations, reads (R) execute queries.

Figure 1: SNB BI's model system and main components.

Comparison of LDBC benchmarks. A comparison of the main LDBC benchmarks is shown in Table 1, including their data model and dynamic behaviour; the type of operations used and choke points stressed (see Section 3.1); and the languages allowed for the implementation. We now provide an overview of these benchmarks.

SNB BI. In this paper, we present SNB BI, the final result of almost a decade of work in designing the Business Intelligence workload of the LDBC Social Network Benchmark, with a first evaluation on multiple systems. As explained in the contributions section, SNB BI is arguably the most refined and challenging analytical (OLAP) database benchmark to date, and targets relational, RDF, and property graph database systems. It is different from purely relational OLAP benchmarks in that the SNB BI queries analyze connection patterns, including thorough complex pattern matching and cheapest path finding between many sources and destinations, sometimes over edges that are not explicitly in the data but computed on-the-fly.

SNB Interactive. LDBC had released the Social Network Benchmark's Interactive workload [20] already in 2015. Although SNB BI shares the graph schema with it, SNB Interactive is a different benchmark aimed at transactional (OLTP) graph management, measuring throughput in sustaining a stream of inserts, while answering also read queries. All read queries start from a single node or a pair of nodes in the graph such that each query visits a limited number of nodes only. Each of these queries is followed by a manifold of simple lookups, that retrieve properties of the visited nodes. This makes the Interactive workload heavier than other transactional benchmarks (e.g. TPC-C [67] or YCSB [15]), but still a far cry from an OLAP benchmark such as TPC-H [68], TPC-DS [69], and SNB BI, in which each of the queries touches a large fraction of a database, which can scale to terabytes of data and beyond.

SNB Graphalytics. At the opposing end of the spectrum, LDBC had also released Graphalytics [31, 32] in 2016, a read-only benchmark that tests graph analytics frameworks, and consists of a suite of graph algorithms (PageRank, clustering, breadth-first search, etc.). Many of these algorithms cannot be expressed efficiently in a query language, and instead are formulated in programming languages such as C++ or Java, embedded through an API in a graph programming framework (such as GraphX [77], GraphBLAS [33]), and typically only analyze the structure of the graph, but not the property values attached to its nodes and edges. SNB BI, on the other hand, targets queries formulated in a domain-specific query language, and like analytical relational database benchmarks do, intensively tests operations on data values (filters, aggregations, top-k, and some value-based joins) in addition to advanced graph pattern matching and path finding. As Table 1 shows, SNB BI thus clearly fills the gap between the transactional SNB Interactive and the algorithmic and read-only Graphalytics benchmark.

Model system. Figure 1a illustrates the model system targeted by SNB BI, implementing the following analytical workflow. Initially, the data resides in a *transactional data system* that serves a business application. To run complex analytical queries, the data set is transferred to an *analytical data system* through an ETL (extract-transform-load) pipeline. To this end, a snapshot of the data is extracted from the transactional system e.g. by creating a *database dump*. This is then cleaned and aggregated to form an *initial snapshot* data set, which is *bulk loaded* to the analytical data system. Subsequent changes in the transactional system are tracked by using its *transaction log*, which is monitored by a *data capture*

system such as Debezium [14] and Oracle GoldenGate [48]. Here, the changes are turned into a stream of *insert* and *delete operations*, applied as *trickle updates*. Meanwhile, the analytical data system serves complex ad-hoc analytical queries to multiple clients.

Contributions and paper structure. To allow benchmark executions that simulate the model system faithfully and yield interpretable results, SNB BI uses several innovative components, e.g. the data and parameter generators, and the benchmark driver (shown in Figure 1b). Moreover, it defines guidelines on how implementations can be audited. The result is a benchmark that clearly advances the state-of-the-art in OLAP and graph DBMS benchmarks. We describe our main contributions in the following sections. Section 2 presents the *SNB BI Datagen* which generates fully dynamic, correlated graphs, and produces batches of update operations. Section 3 outlines the *query template design process* and highlights example graph-shaped queries. Section 4 describes *parameter curation* techniques that ensure stable runtimes. Section 5 defines the *benchmark's workflow*, its balanced scoring metrics and key *auditing guidelines*. Section 6 shows experimental results with two systems under test, the Umbra RDBMS [43] and the TigerGraph graph DBMS (GDBMS) [17]. We end the paper by reviewing related literature in Section 7 and summarizing our findings in Section 8.

2 DATA SET AND BATCH GENERATION

The SNB Datagen generates social network graphs for the SNB benchmarks. The key design goals of Datagen are to (1) produce dynamic social networks with *realistic* characteristics (network structure, attribute distributions, and correlations), (2) *scale* for networks with 100B+ edges, and (3) ensure *reproducibility*.

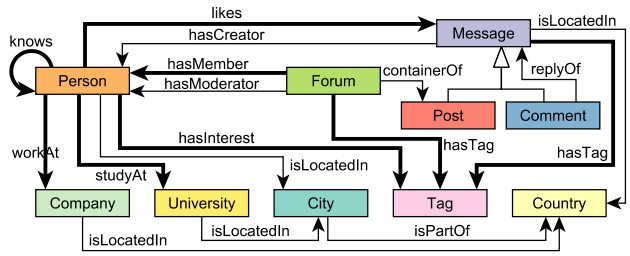


Figure 2: Schema of the social network graph. Many-to-many edges are highlighted with thick lines. Attributes are omitted.

2.1 Conceptual Overview

Schema. The LDBC SNB uses a graph schema with 14 node types connected by 20 edge types. The data set consists of a Person–knows–Person graph and a number of Message trees within Forums. Messages are connected to Persons by creatorship and likes. A simplified schema is shown in Figure 2.

Correlated graph. The degree distribution of the Person–knows–Person network is modelled after Facebook, with the social graph exhibiting the *small-world phenomenon* [73] characterized by a small diameter. Property-value (a.k.a. column-value) distributions are skewed. Completely unique w.r.t. other graph generators, column values *correlate* within an entity (e.g. French people have predominantly French names) but also correlate with *structural features* of the graph. For instance, following the *homophily principle* [38], people are more likely to be friends if they studied at the same school at the same time, live in close proximity, and/or have the same interests. The queries in the SNB BI workload exploit these correlations, and for some queries with two parameters, two variants exist. This allows us to pick two structurally correlated parameter values for one query variant, while picking anti-correlated values for the other (Section 4.3). This results in a query optimizer seeing structurally identical query variants, with similar-looking base table selectivities that will, however, have completely different join hit rates.

Dynamic graph. The SNB social network is a dynamic graph, where friendships and messages are exchanged at realistic time intervals. However, these intervals may be skewed to represent *flashmob* events where discussed topics spike in popularity for a period before decaying back to normal [36]. A certain query variant may be affected by a flashmob, resulting in unexpectedly large cardinalities, while the other variant is not.

2.2 Generating Deletes

Delete operations. Generating complex delete operations is an important and unique feature of the SNB BI Datagen. We argue that supporting efficient delete operations is a crucial and challenging feature in analytical data sytems: (1) Applying the *trickle updates* (Figure 1a) necessitates efficient deletions. (2) Delete operations are required in order to comply with privacy laws, e.g. the European Union's General Data Protection Regulation (GDPR), which has shown to have a significant impact on the performance of data systems [61]. (3) Deletion operations *limit the algorithms and data structures* that can be used by a system. For instance, many incremental computations (e.g. for shortest path) are significantly more challenging in the presence of deletes [53], and several graph-oriented storage formats support efficient insertions but not deletions [12].

Temporal graph with lifespan attributes. To produce dynamic graphs which include deletions, the SNB BI Datagen first produces a *temporal graph*, which contains all entities that exist at some point during the simulation period. Within this, entities are assigned *lifespan attributes*, i.e. their *creation date* and *deletion date*. For entities to have valid lifespans: (1) updates must follow a logical order, e.g. two Persons can become friends (knows edge) only after both join the network and before either Person leaves the network; (2) updates in the graph should not violate the cardinality prescribed by the schema, e.g. a Forum can only have a single moderator (hasModerator edge); (3) updates need to satisfy the semantic constraints required by the application domain, e.g. a Person can only create a Post in a Forum if they are a member of the Forum (hasMember edge). To ensure valid lifespans, we define intervals in which creation and deletion events can logically occur. Our techniques for deriving these are presented in [75]. The intervals are defined during the time period starting at *simulation start* (the time when the simulated social network is created) and ending at the *network collapse* (the time when the social network is assumed to be shut down) [37]. The period used in the benchmark starts at *simulation start* and ends at *simulation end* (a point in time before the network collapse).

Cascading deletes. Deleting graph entities often triggers a *cascading delete*. Firstly, it is required there are no dangling edges thus nodes must be always deleted with all their edges. Secondly, node deletions can also trigger the deletion of other nodes, e.g. the deletion of a Forum implies the deletion of all its Posts and their Comments.

Deletion probabilities. To maintain Datagen's design principle of realism, deletion probabilities and timestamp distributions were derived from real-world data where possible, namely for Person nodes [37], knows edges [41], and likes edges [3]. The deletion of Forum nodes (1% during the simulation period) and hasMember edges (5%) are not motivated by empirical evidence. The deletion probability of Persons is determined as follows. People with more connections are less likely to leave a social network to avoid losing social capital they have accumulated [37]. In Datagen, when a Person is generated, the total number of knows connections they will make across the simulation period is determined. This information is leveraged using the deletion distributions provided in [37], whose authors report a temporal analysis of the now-defunct Hungarian social network, iWiW. It is assumed the arrival rate of Person nodes into the network is constant over the simulation period, thus the same is assumed of the departure rate with deletion timestamps selected uniformly from the valid deletion interval for a Person.

2.3 Initial Snapshot and Updates

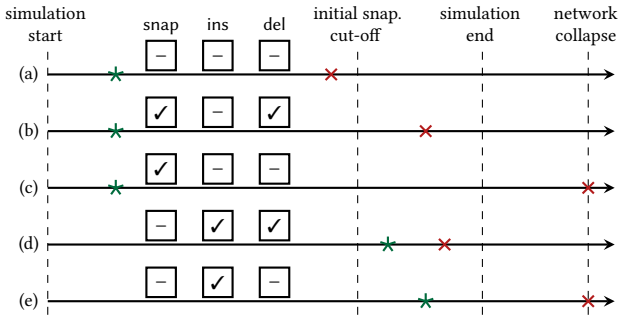


Figure 3: Possible dynamic entity *creation* ✶ and *deletion* × dates with respect to simulation start, initial snapshot cut-off, simulation end, and network collapse. We indicate whether the entity is included in the initial *snap*shot, the *in*sert updates, and the *del*ete updates.

To produce a separate initial data set and a stream of trickle updates (shown in Figure 1a), we establish a *cut-off date* and associate the entities with one or more of the following categories: (1) the *initial snapshot*, which contains entities before the cut-off date; (2) entities *inserted* after the cut-off date; (3) entities *deleted* after the cut-off date. An entity may occur in multiple of these categories depending on their creation date and deletion date. Figure 3 enumerates all possibilities: In case (a), the entity is created and deleted before the cut-off, therefore it is discarded from the data set. (b) presents the case in which the dynamic entity is created before the cut-off, deleted after the cut-off, but before the simulation end. Such an entity is serialized into the initial snapshot and spawns a delete operation. In case (c), the entity is created before the cut-off, but is deleted after the simulation end, resulting only in inclusion in the initial snapshot. In case (d), the entity occurs after the initial snapshot cut-off and is deleted before the simulation end, therefore it spawns an insert and delete operation. Lastly, in case (e) the entity is created after the initial snapshot but is deleted sometime after the simulation end and thus spawns only an insert operation.

Batch splitter. The *batch splitter* (Figure 1b) produces update batches consisting of insert operations (Section A.2) and delete operations (Section A.3), each capturing the updates for a time period of one day. Note that if an entity is created and deleted within the period of the batch, it will be included in both update operations. The rationales behind this decision are to (1) follow the behaviour of a typical model system (Figure 1a), where handling these redundant operations would be the task of the analytical data system and (2) allow modelling of streaming updates where there is no batching.

2.4 Data Sets

Formats and layout. The SNB BI Datagen is capable of serializing CSV (Comma Separated Value) files with different layouts, depending on whether edges with a cardinality of one-to-many are merged in the node files as foreign keys (merged-FK) or projected into separate files (projected-FK). Datagen is also capable of generating Parquet files, a format popular for storing *data lakes* [29].

Table 2: SNB BI data sets. *k:* thousand, *M:* million, *B:* billion.

	SF10	SF30	SF100	SF300	SF1,000	SF3,000	SF10,000
#nodes	27M	78M	255M	738M	2.4B	7.2B	23B
#edges	170M	506M	1.7B	5.1B	17B	51.9B	173B
#Person	68k	170k	473k	1.2M	3.4M	9M	26M
#knows	1.8M	5.5M	19M	55.7M	187M	559M	1.9B
#insert ops	44.6M	127M	399M	1.1B	3.3B	8.9B	27B
#delete ops	353k	1M	3.3M	9.3M	28.9M	79.7M	245M

Scale factors. Similarly to the TPC benchmarks [68, 69], SNB BI's data sets are characterized by their scale factors (SF). This is defined as the overall size of the uncompressed CSVs in merged-FK format, measured in GiB, including both the initial snapshot and the update operations. Table 2 shows the key properties of the data sets.

2.5 Scalability and Reproducibility

Scalability. The old version of the SNB Datagen used Hadoop [49] and was limited to generating SF1,000 (1,000 GiB) data sets. The SNB BI Datagen was ported to Spark to improve scalability. It supports both on-premise Spark clusters and cloud services such as AWS Elastic MapReduce (EMR). Due to space constraints, we refrain from a detailed analysis of Datagen's scalability and refer the reader to our papers [20, 49] and blog posts [62, 63].

Reproducibility. The Datagen guarantees reproducibility, i.e. for a given configuration, it always produces the same graph regardless of the number of machines it is executed on. This is achieved by using a deterministic blocking algorithm where Persons are divided into blocks of 10k, with each block having its own independent state and only depending on the block id. Blocks can then be deterministically sorted during the edge generation and edges are merged to

Table 3: Challenging graph-related choke points featured in the read queries. *Notation:* ⊗ covered, ○ not covered.

	CP	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20
challenging joins	CP-2.x	○	⊗	⊗	⊗	⊗	⊗	⊗	⊗	○	⊗	⊗	⊗	⊗	⊗	○	⊗	⊗	⊗	○	○
cheapest path finding	CP-7.6	○	○	○	○	○	○	○	○	○	○	○	○	○	○	⊗	○	○	○	⊗	⊗

eliminate duplicates. The pseudorandom number generators used during the generation are seeded with constant values.

3 READ QUERY TEMPLATES

SNB BI consists of 20 parameterized *read query templates*, herein referred to as *queries*. These search for graph patterns (often implying join-heavy operations on many-to-many edges), traverse hierarchies, and compute cheapest paths (a.k.a. weighted shortest paths). Additionally, they include filtering, grouping, aggregation, and sorting operators. While all queries explore a large portion of the graph, they only return the top-k (typically 20 or 100) results, keeping their result sizes compact to avoid emphasizing the client-server network protocol's role in the benchmark [51].

Query languages. To ensure *portability*, the queries are defined in plain text and implementations are not required to use a particular query language. At the same time, we designed the queries such that they will be expressible in the upcoming ISO standard SQL/PGQ and GQL query languages [16][1].

3.1 Choke Point-Based Design Methodology

LDBC's query design process relies on using *choke points* [11, 19], i.e. challenging aspects of query processing. SNB BI includes 38 choke points divided into 9 categories: aggregation performance, join performance, data access locality, expression calculation, correlated subqueries, parallelism and concurrency, graph specifics, language features, and update operations. A full breakdown of the choke points and their coverage is given in [6, Appendix A]. Here, we focus on the join and graph-specific choke points. Their coverage is shown in Table 3, which shows that 18 of the 20 read queries cover at least one choke point related to joins or path finding.

Explosive and redundant multi-joins. In recent years it has become clear that graph pattern matching, or equivalent multi-join queries over many-to-many relationships, typically generate very large intermediate results when executed with traditional join algorithms. This is especially the case for cyclical join graphs (corresponding to cyclic graph queries). It was proven in theory [46] and shown in practice [21, 39, 72] that "worst-case optimal" *multi*-join algorithms can avoid these large intermediates and outperform traditional joins. Following this, there has been increased attention on *redundancy* in join results for acyclic subgraph queries (even when produced by worst-case optimal joins), which can be eliminated using *factorized* query processing techniques [9, 28, 47]. Graph pattern matching queries that contain large join patterns will trigger these phenomena, and SNB BI is the first OLAP benchmark to include these.

Expressive path finding. SNB BI contains queries that require an efficient implementation of cheapest path finding between many pairs. Expressing such queries requires a query language which supports either path finding or recursion. The underlying system implementation must then handle this with an optimized execution strategy, as recursing to try all paths will not scale. As some of this path finding includes on-the-fly computed edges (joins) between nodes, the queries can benefit from *path expressions*, as proposed in Oracle's PGQL language [71] and as part of the upcoming GQL and SQL/PGQ languages. The path finding required by SNB BI not only tests connectivity (as supported in SPARQL), but also requires returning the *cheapest cost* along weighted paths (necessitating SPARQL extensions [40]).

3.2 Example Queries

In order to defeat trivializing complex query performance by caching, SNB BI instantiates the parameterized query templates with different *substitution parameters* (a.k.a. parameter bindings). In this section, we describe and analyze four read queries while the rest of them are given in Section A.1. We denote the parameters with the $ symbol and discuss their generation in Section 4.

3.2.1 Q11: Friend triangles (Figure 4a). For a given Country `$c`, count all the distinct sets of Persons living in Cities of `$c`, who form a triangle: `p1` knows `p2`, `p2` knows `p3`, `p3` knows `p1`, and these edges were created in the interval [`$startDate`, `$endDate`].

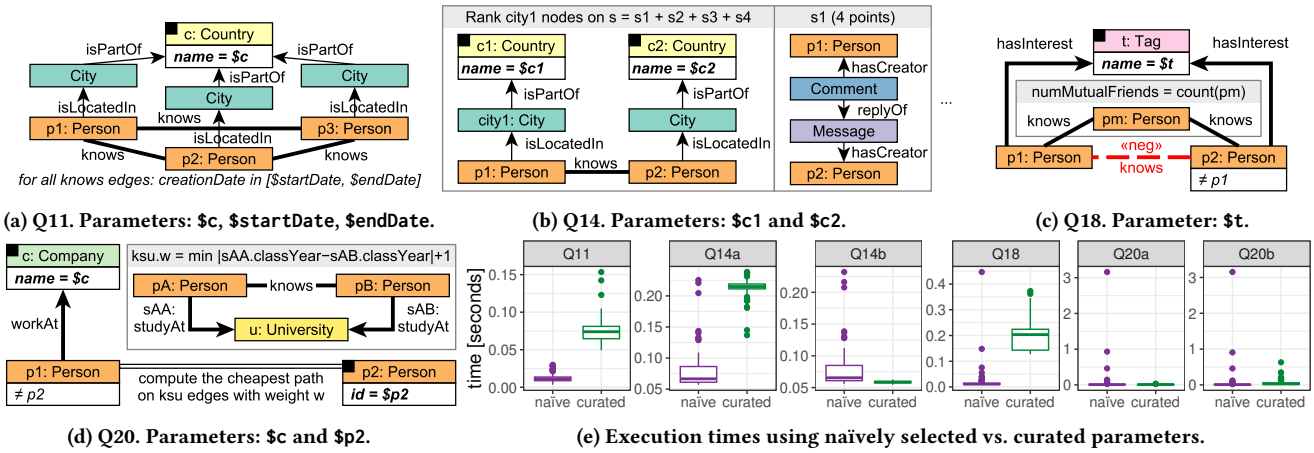
Analysis. This query imposes two key difficulties. First, systems should efficiently filter the knows edges based on the location of their endpoint Persons (Cities in Country `$c`) and the date range. Second, given a large number of knows edges even after filtering, efficient enumeration of `p1`–`p2`–`p3` triangles (a cyclic subgraph query) requires worst-case optimal multi-joins.

3.2.2 Q14: International dialog (Figure 4b). Given two Countries `$c1` and `$c2`, consider all pairs of Persons `p1`, `p2` such that they know each other, `p1` is located in `$c1`, and `p2` is in `$c2`. Score them based on the volume of their interactions through Messages (details omitted). For each City in `$c1`, return the highest scoring pair.

Analysis. The optimal query plans for this query are different based on whether Countries `$c1` and `$c2` are correlated or anti-correlated (Section 4.3). For the ranking, *top-k pushdown* can be exploited: once a result for a City in `$c1` is obtained, extra restrictions in a selection can be added based on the value of this element. As the score of two Persons does not depend on any query parameters, precomputing it as an attribute on the knows edge can be beneficial.

3.2.3 Q18: Friend recommendation (Figure 4c). For each Person `p1` interested in Tag `$t`, recommend new friends (`p2`) who do not yet know `p1`, have at least one mutual friend (`pm`) with `p1`, and are also interested in `$t`; rank based on the number of mutual friends.

[1] With the "cheapest path" language opportunity fulfilled



(a) Q11. Parameters: $c, $startDate, $endDate.

(b) Q14. Parameters: $c1 and $c2.

(c) Q18. Parameter: $t.

(d) Q20. Parameters: $c and $p2.

(e) Execution times using naïvely selected vs. curated parameters.

Figure 4: Read query templates discussed in this paper. Thick lines denote many-to-many edges, red dashed lines imply negative conditions. The black square ■ in a node's top left corner denotes that the node is uniquely specified by a query parameter.

Analysis. This query is inspired by Twitter's recommendation algorithm [27]. Implementations of this query can exploit factorization: systems can count the number of mutual friends without explicitly enumerating all (p1, pm, p2) tuples.

3.2.4 Q20: Recruitment (Figure 4d). Construct a graph from the knows edges where the endpoint Persons pA and pB have attended at least one University together. For these Universities, we consider the pairs of studyAt_A and studyAt_B edges for pA and pB, respectively, and define the weight of the corresponding knows edge as follows:

$$w = \min_{\text{studyAt}_A,\text{studyAt}_B} |\text{studyAt}_A.\text{classYear} - \text{studyAt}_B.\text{classYear}| + 1$$

Then, given a Company $c and a Person $p2, find a Person p1 ≠ $p2, who works at $c and has the cheapest path to $p2.

Analysis. This query performs *graph projection* [7]. Instead of materializing this graph in the database, systems may represent it using a compact in-memory structure such as CSR (Compressed Sparse Row) [55]. To perform the cheapest path computation, a single-source cheapest path algorithm (starting from $p2), such as Dijkstra's algorithm, can be used. As the projected graph is independent of query parameters, precomputing it can be beneficial.

4 PARAMETER CURATION

4.1 The Need for Parameter Curation

A disadvantage of executing the same read query template with different parameters is that the intermediate results and runtimes can be severely influenced by the parameter values. This is particularly the case in SNB BI with its explosive joins, skewed out-degrees, skewed value distributions, correlated value distributions, and structural correlations. Moreover, the updates (including cascading deletes) can significantly change the portion of the graph reached by the same query executed at different times. In order to keep query performance understandable we need to actively *curate* parameters, such that different parameters executed at different logical times still lead to stable and, therefore, understandable results. We achieve this through *parameter curation* [20, 25], a data mining process of looking for parameter values with suitably similar characteristics. Our approach improves on existing parameter curation techniques by improving their scalability as well as increasing the maintainability of the parameter generator code.

The effect of naïve parameter selection. We demonstrate the importance of parameter curation with the Umbra system using the environment and implementation described in Section 6.3. For this experiment, we ran a naïve parameter generator which selects parameters following a uniform distribution. We loaded the SF100 data set in the database and ran 100 query instances (sequentially) of Q11, Q14*a*, Q14*b*, Q18, Q20*a*, and Q20*b*, with both naïve and curated parameters, and measured their individual execution times (Figure 4e). The results show that naïvely selected parameters include numerous *outliers*, which often take more than an order of magnitude longer to finish than the median query execution time, while the curated variants contain fewer outliers.

4.2 Parameter Generation Steps

Our parameter curation process is a two-step process: we first generate *factors*, then used them for generating *parameters*. The factor and the parameter generator components (Figure 1b) are executed for each scale factor and are independent of the serialization format/layout of the data set.

Factor generation. The factor generator produces *factor tables* containing data cube-like summary statistics [24] of the temporal graph, e.g. the number of Persons per City or the number of Messages per day for each Tag. It also contains tables derived with more complex computations (e.g. connected components), see Section 4.3.

Parameter generation. To find suitable substitution parameters that (presumably) lead to the same amount of data access and thus similar runtimes, we first identify the factor table containing the summary statistics of the query's parameters. For example, the template of Q14 (Figure 4b) uses the parameters Country $c1 and Country $c2. Therefore, we use the `countryPairsNumFriends` factor table which contains $c1, $c2 pairs and the number of friendships where

one Person lives in $c1 and the other lives in $c2. Using this table, we select the pth percentile from the distribution as the *anchor*, then rank the rest of the distribution based on their absolute difference from the anchor and take the top-k values. We shuffle the values using a hash function to avoid introducing artificial locality, where e.g. subsequent queries start in nodes from the same ID range. Listing 1 shows the plain SQL parameter generation query for Q14*a*.

```
SELECT c1, c2 FROM (
  SELECT c1, c2, abs(frequency - (
    SELECT percentile_disc(0.98) WITHIN GROUP (ORDER BY frequency)
      AS anchor FROM countryPairsNumFriends
    )) AS diff FROM countryPairsNumFriends ORDER BY diff, c1, c2
) ORDER BY md5(concat(c1, c2)) LIMIT 50
```

Listing 1: Parameter generation SQL query for Q14*a*.

4.3 Parameter Curation for Graph Queries

We discuss two parameter curation cases that are particularly important in graph data management.

Correlated vs. anti-correlated parameters. Our parameter curation provides a straightforward way of selecting start entities which are affected by (structural or attribute-level) correlation vs. anti-correlation: corresponding parameters can be found by selecting a high vs. low percentile as the anchor in the parameter generation query. For example, for Q14 (Figure 4b) we set variant a to $p = 0.98$ (correlated) and variant b to $p = 0.03$ (anti-correlated). Figure 4e and Table 4 show the runtimes of these variants.

Reachable vs. unreachable node pairs. Queries Q15, Q19, Q20 include cheapest path finding queries computed between given (sets of) Persons. These queries are particularly challenging for parameter curation as query runtimes may significantly differ based on the existence of a path between the selected endpoints. Moreover, the presence of a path between two nodes *at a given time* does not guarantee that it will always exist during the benchmark execution as deletions can render the endpoints of a path unreachable.

The problem of selecting reachable vs. unreachable node pairs is the most pronounced for Q20 (Figure 4d), which looks for cheapest paths in a sparse subgraph consisting of knows edges between Persons who in the original graph "know each other and attended the same University" (ksu). Therefore, this query has two variants based on whether Person $p2 is reachable from at least one Person p1 who is an employee of $c. For Q20*a*, it is guaranteed that no such path exists, while for Q20*b*, it is guaranteed that there is a two-hop path. To allow generating such parameters, during the *factor generation*, we use the *temporal graph* to compute the ksu factor table, i.e. pairs of Persons who attended the same University. As the temporal graph contains all edges that exist in the graph at any time, this is an *overapproximation* of the ksu edges that are available at a given time during the benchmark period (after the cut-off date and before the simulation end). We then run a *connected components* algorithm on the graph of the ksu edges and save its results to the factor table ksu_components. Additionally, we also return compact views of the knows, studyAt, and workAt edges which only contain their source ID, destination ID, and temporal attributes (creation date, deletion date).

We perform the parameter generation as follows. For Q20*a*, we select Person pairs from the ksu_components table who are in different components. Due to the overapproximation, it is guaranteed that if two nodes are in different components in the temporal graph, they will be in different components at any given time in the actual graph during the simulation. For Q20*b*, we perform the following steps. (1) We use the creation date and deletion date attributes of knows and studyAt edges to create the ksu_filtered table by only keeping edges that exist during the entirety of the benchmark period. (2) We create a set of Company candidates based on the number of employees using the technique shown in Listing 1. (3) For each Company, we get their employees using the temporal attributes of the workAt edges (to ensure they work at the Company during the benchmark period) and perform two joins on ksu_filtered to compute two-hop paths to get $p2 candidates. (4) We return a deterministic sample using a hash-based shuffling.

4.4 Query Variants

12 queries have a single variant, while 8 queries have two variants, yielding a total of 28 query variants. Variants of Q2, Q8, Q16 are parametrized with a flashmob vs. a non-flashmob date (Section 2.1). Variants of Q14 and Q19 select correlated vs. non-correlated Countries/Cities. Q10's variants differ in the degree of the start Person (high vs. low), while Q15's variants have different path lengths and time intervals (4 hops and one week vs. 2 hops and one month). Q20 variants differ on whether a path exists between the two endpoints.

4.5 Scalability and Reproducibility

Scalability. The *factor generator* is part of the SNB Datagen and runs after the *temporal graph* has been created. It is implemented using the Spark DataFrame API for distributed execution. While its data cube-like computations use expensive, aggregration-heavy queries, the derived factor tables are *compact*, e.g. SF10,000 has only 20 GiB of factors in compressed Parquet format, the equivalent of approximately 100 GiB in CSV format, i.e. 1% of the total data set size. The *parameter generator* queries are executed in DuckDB [52]. To demonstrate that our setup is scalable, we report runtimes for the parameter curation on the largest data set, SF10,000: the parameter generation took 22.2 minutes on an r6id.32xlarge instance in AWS EC2 (128 Intel Xeon 8375C 2.90 GHz vCPU cores, 1 TiB RAM).

Reproducibility. It is important to guarantee that the parameter curation process is reproducible. To this end, we leverage that the graph generator and, consequently, the factor generator are reproducible. To ensure that the parameter generation queries yield deterministic results we define a total ordering in each query. To provide deterministic shuffling we base the ordering on MD5 hashes of concatenated attribute values, see Listing 1 for an example.

5 BENCHMARK WORKFLOW AND AUDITING

5.1 Benchmark Phases and Workflow

SNB BI's execution includes three distinct phases (Figure 1b): the *load*, the *power batch*, and a sequence of *throughput batches*. Both type of batches consist of writes corresponding to one day of simulation time in the social network, and 30 instances for each of the 28 read query variants (Section 4.4).

Load. To begin the benchmark, an initial snapshot (Figure 3) is loaded into the system under test (SUT), timed as t_{load}.

Power batch. The workflow includes a single *power batch*. This first performs the write operations, followed by the reads with no concurrency allowed between any query instances. The following times are measured: the runtime of applying the writes (w), the time spent on reads by each query variant, denoted as $q_1, q_{2a}, q_{2b}, \ldots, q_{20a}, q_{20b}$; and the total time spent on reads. The sequential execution implies that achieving good performance during this batch necessitates *intra-query parallelism* from the SUT.

Throughput batches. The *throughput batches* run the same operations as the power batches but allow fully concurrent execution to exploit *inter-query parallelism*. To qualify for a valid run, experiments shall run $n_{throughput\ batches} \geq 1$ throughput batches for a timespan of $t_{throughput\ batches} \geq 1$ hour.

Concurrent vs. disjoint RWs. Implementations can choose to run either *concurrent* or *disjoint read-write (RW) operations* in their throughput batches. A concurrent RW setup allows for the interspersing of update operations among read queries. Under disjoint RW operations systems should first perform all write operations, followed by (potentially concurrent) reads. The rationale behind this design choice is to enable the participation of *read-optimized data analytical systems* such as Spark [78]. In the long term, however we expect the top SNB BI implementations to choose a concurrent RW setup as this potentially leads to improved resource utilization.

5.2 Scoring Metrics

SNB BI provides four scoring metrics: the *power score*, the *throughput score*, and their price-adjusted variants, the *per-$ power score* and the *per-$ throughput score*.

Power score. The definition of SNB BI's power score follows TPC-H in using a geometric mean, ensuring that there is an incentive to improve all queries, no matter their running time. Its formula uses the execution times during the power batch, measured in seconds:

$$power@SF = \frac{3{,}600}{\sqrt[29]{w \cdot q_1 \cdot q_{2a} \cdot q_{2b} \cdot \ldots \cdot q_{20a} \cdot q_{20b}}} \cdot SF$$

Throughput score. The throughput score is based on t_{load}, measured in hours, the cumulative execution time and the number of the throughput batches executed. The subtraction of t_{load} ensures that the scoring rewards systems with efficient bulk loaders (unlike in TPC-H/DS which do not include load performance in their metrics):

$$throughput@SF = (24\text{ hours} - t_{load}) \cdot \frac{n_{throughput\ batches}}{t_{throughput\ batches}} \cdot SF$$

Price-adjusted scores. We follow TPC's specification for reporting prices [70]. The price is established as the *total cost of ownership (TCO)* for the SUT used in the benchmark, reported as a breakdown of machine cost, software license costs, and maintenance costs for 3 years. For cloud deployments, the cost of a 3-year reserved instance should be used. To determine the price-adjusted scores, we factor in the *TCO* by multiplying the respective base score by $\frac{1{,}000}{TCO}$.

5.3 Auditing Rules

LDBC defines stringent auditing rules for its benchmarks [6, Chapter 7]. Here, we summarize two key rules for SNB BI.

ACID test. Systems running *concurrent RWs* in throughput batches must pass the LDBC ACID test suite [74] to ensure that they comply with their claimed isolation level (*snapshot isolation* or higher is required). Systems *running disjoint RWs* can omit this test as their execution does not require transactional isolation for correctness.

Query languages. Implementations are only allowed to use *domain-specific query languages* to express the SNB BI read queries and update operations. This allows the use of languages with imperative traits such as Gremlin [54] and GSQL but forbids using general-purpose programming languages and accessing the data through a low-level API. This criterium is different from SNB Interactive where API-based implementations are also allowed (see Table 1).

6 PERFORMANCE EXPERIMENTS

We conducted a series of experiments to demonstrate that SNB BI satisfies key criteria of benchmark design [22, 30]. Its *portability* is confirmed by providing three complete implementations. Its *scalability* is proven by running experiments on data sets up to SF10,000 (i.e. 10,000 GiB). Finally, its *economic viability* is shown through a detailed cost breakdown of our experiments.

6.1 Disclaimer

The experiments presented here are not LDBC Benchmark Results, as they were not *audited*, do not include price-adjusted scores, and cannot be used for product comparison. Instead, they were conducted as part of LDBC's *standard-establishing audit process*, which requires a complete, specification-compliant execution of the benchmark on two systems and makes subsequent official audits possible [35]. To counter marketing misuse, LDBC trademarked the term "LDBC Benchmark Result", and stipulates that it can only be used to describe the outcome of experiments that have passed an official LDBC audit. Academics are encouraged to use LDBC benchmark technology, also without audits, provided their paper states that the results are not LDBC Benchmark Results (regardless of the capitalization of these three words or the presence of a trademark logo). For such usage, LDBC also encourages describing those aspects where such a non-audited result deviates from the benchmark implementation rules (e.g. only executing a power batch, or executing a power batch without the write operations, or exploiting concurrency without the required ACID properties, or using non-official parameter bindings, or using incomplete – or different – data, queries, or writes).

6.2 Systems Under Test

We created three implementations of the workload in Neo4j[2], Umbra [43], and TigerGraph [17]. We used the Neo4j implementation for cross-validation at SF10, and report performance results from Umbra and TigerGraph. In this section we discuss the latter two implementations, describing their schema and their approach for handling the initial bulk load, read queries, and update operations.

[2] https://neo4j.com/ (accessed on Dec 16, 2022)

Table 4: Benchmark results for Umbra and TigerGraph on SF30 to SF10,000: scoring metrics, load times, power test results (write batch runtime to *average query runtimes* for each query variant), throughput test statistics, and experiment costs. Times are reported in seconds.

	Umbra			TigerGraph	
	SF30	**SF100**	**SF300**	**SF1,000**	**SF10,000**
power@SF	75,761.75	103,308.45	110,473.72	17,821.02	61,319.43
throughput@SF	n/a	28,996.42	26,251.13	7,655.88	23,132.08
load time	68.70	211.92	668.81	4,786.00	6,321.00
power batch writes	6.79	14.66	45.66	1,369.52	3,272.27
power writes w/o prec.	1.48	2.84	8.01	445.37	859.48
Q4 precomputation	0.68	1.67	4.66	126.00	336.96
Q6 precomputation	0.31	1.03	2.96	155.58	491.11
Q19 precomputation	2.10	7.38	25.60	615.07	1,486.82
Q20 precomputation	2.21	1.75	4.42	27.51	97.90
total precomputation	5.31	11.82	37.65	924.16	2,412.79
power batch reads	84.14	243.47	945.70	8,879.12	25,423.65
Q1	0.02	0.05	0.16	9.40	10.54
Q2*a*	0.04	0.10	0.32	9.56	25.29
Q2*b*	0.02	0.04	0.09	2.55	12.04
Q3	0.06	0.16	0.71	5.85	23.83
Q4	0.03	0.07	0.17	2.16	5.88
Q5	0.04	0.09	0.31	1.63	8.13
Q6	0.03	0.07	0.24	1.82	9.03
Q7	0.04	0.10	0.25	4.02	26.29
Q8*a*	0.02	0.05	0.12	3.55	10.50
Q8*b*	0.02	0.05	0.15	1.68	6.11
Q9	0.19	0.55	1.38	12.16	25.18
Q10*a*	0.09	0.31	0.79	11.93	30.23
Q10*b*	0.05	0.14	0.49	5.33	12.18
Q11	0.04	0.09	0.19	7.15	13.65
Q12	0.03	0.09	0.27	10.16	20.34
Q13	0.05	0.13	0.33	35.20	75.27
Q14*a*	0.07	0.22	0.69	13.01	36.73
Q14*b*	0.02	0.06	0.17	5.10	16.64
Q15*a*	0.34	0.58	1.74	25.16	65.92
Q15*b*	1.28	4.36	20.66	67.31	159.06
Q16*a*	0.06	0.14	0.33	7.98	40.56
Q16*b*	0.05	0.12	0.30	2.72	8.41
Q17	0.06	0.18	0.43	6.82	27.23
Q18	0.07	0.21	0.57	27.94	141.04
Q19*a*	0.02	0.03	0.07	6.08	14.44
Q19*b*	0.02	0.03	0.07	6.05	15.80
Q20*a*	0.01	0.01	0.01	1.99	3.53
Q20*b*	0.02	0.08	0.50	1.66	3.59
$n_{throughput\ batches}$	32 batches	13 batches	4 batches	1 batch	1 batch
$t_{throughput\ batches}$	3,242.52	3,864.08	3,918.97	10,660.30	34,618.16
total execution time	3,333.71	4,122.39	4,910.60	20,908.95	63,314.11
experiment cost	$18.79	$21.26	$24.34	$66.75	$1,849.97

Umbra. Umbra is a mostly PostgreSQL-compatible hybrid OLTP & OLAP research RDBMS [43] with compiled query execution [34, 42], and support for worst-case optimal multi-joins [21]. The read queries, initial bulk loading, and update operations are implemented in SQL in the PostgreSQL dialect. The read queries are verified to work on PostgreSQL, and as they require no proprietary SQL extensions, they could be adapted to work on most RDBMSs. Even though the queries only use plain SQL, their efficient execution relies on sophisticated query optimization techniques such as unnesting arbitrary queries [44] and cost-based join ordering [45].

The relational schema is designed to eliminate as many tree traversals of the data as possible during the read queries. For instance, every Comment is stored with the ID of its corresponding root Post. This allows Umbra to omit the expensive tree traversal to find the root Post for a given Comment. Storing this additional ID with every Comment has nominal storage costs, and is relatively easy to maintain as there are no cut-and-link operations; a Comment is never moved, it is only inserted (below an existing Message with an existing root Post) and deleted (which results in child Comments being deleted). As Umbra does not yet support automatic cascading deletes, the set of tuples to be deleted is initially computed manually in auxiliary tables then deleted with the SQL `DELETE USING` clause.

TigerGraph. TigerGraph is a distributed massively parallel processing (MPP) GDBMS using a native graph storage format. It is designed for handling HTAP query workloads. It offers GSQL, a Turing-complete query language [17] which provides both declarative features (e.g. graph patterns) as well as imperative ones (e.g. for expressing iterative graph algorithms with loops and accumulators primitives) [18]. GSQL allows users to program at a high abstraction level and select a (close to) optimal join ordering at the same time.

The SNB BI graph schema is defined using GSQL DDL, which consists of node types, edge types and graph type definition statements. The initial bulk loading, and the subsequent insert and delete operations are implemented using the GSQL data loading language (DLL). When TigerGraph is deployed on a cluster of machines the input data is distributed across the cluster and loaded concurrently. Deletes are implemented with GSQL queries which search for the entities affected by a given delete operation, then delete them.

The read queries are implemented in GSQL and the compiled executions are distributed across the cluster in an MPP fashion. Accumulators [18] are used to perform aggregation and filtering. Common static computations were identified and were formulated as GSQL queries whose results are materialized as auxiliary attributes to speed up Q4, Q6, Q14 and Q19 (computed together and reported under Q19's precomputation time), and finally Q20. These attributes require some extra space, but significantly reduce the cost over multiple runs. For example, the member counts per Country of a Forum are precomputed and the largest member count is materialized in the Forum node. When Q4 is evaluated, the largest member count is used directly to calculate the top-k popular Forums.

6.3 Benchmark Environment and Execution

We describe the benchmark environments, data sets, and the workflow used for conducting the experiments.

Umbra. We used Umbra version `cbad59200`, running in a container using Docker 20.10. The experiments were run in AWS EC2 on an `m6id.32xlarge` instance (128 Intel Xeon 8375C 2.90 GHz vCPU cores, 512 GiB RAM, 4×1.9 TiB NVMe SSDs in RAID0 configuration) running Ubuntu 22.04 as the host OS and Ubuntu 22.10 in the container. The data sets (SF30, SF100, SF300) were serialized as uncompressed CSV files in the `merged-FK` layout.

TigerGraph. We used TigerGraph version 3.7.0. The experiments were run in the GCP Compute Engine in a distributed setup, using `n2d-highmem-32` instances (32 AMD EPYC 7B12 2.25 GHz vCPU cores, 700 GiB non-NVMe SSD storage, 256 GiB RAM) with

CentOS 7 v20220406. The SF1,000 experiment used 4 instances while the SF10,000 experiment was performed on 48 instances. The data sets were serialized as CSV files in the projected-FK layout.

Workflow. The experiments were executed according to the workflow given in Section 5.1. Both implementations opted to run *disjoint RWs* in their throughput batches and chose not to exploit inter-query parallelism due to the high memory consumption of individual queries. Instead, they used the same code path for their power batch and throughput batches.

6.4 Analysis of the Results

We summarize our main findings from the results shown in Table 4.

Portability and scalability. The existence of three feature-complete SNB BI implementations (Neo4j, TigerGraph, Umbra) illustrates the benchmark's portability. The benchmark is also scalable: its data and parameter generators can produce inputs for SF10,000 experiments and its driver can stress DBMSs at this scale.

Tractable complexity. The results confirm that the read queries and update operations issued by SNB BI are *tractable*, i.e. they can be evaluated in a reasonable time and it is possible to perform a complete benchmark execution in a few hours on large scale factors.

Significant performance differences for 3 query variants. The effect of query variants clearly shows for Q2, Q10, Q14: the *a* variants are significantly slower than the *b* variants, as expected. For Q8 and Q16, this can only be observed for TigerGraph.

Limited performance differences for path queries. For the path queries, Q15, Q19, and Q20, the performance differences between variants are limited. In Q15's case, the selected time interval is the dominant factor, making variant *a* simpler. In Q19's case, the complexity of cheapest path finding between correlated vs. anti-correlated Cities shows little difference, less than 10%. Interestingly, Umbra achieves 2–42× faster runtimes on Q20*a* compared to Q20*b* as it can quickly identify when there is no path between the nodes by traversing the smaller connected component from one endpoint.

High performance from Umbra. Umbra demonstrates high performance thanks to its state-of-the-art architecture, execution model, and sophisticated optimizer. On SF30, Umbra runs out of throughput batches to evaluate, causing the throughput batch execution time to be less than the required time (1 hour), therefore the throughput@SF score cannot be determined on this scale factor. The performance of Umbra on path finding queries (Q15, Q19, and Q20) can be particularly surprising as path finding is difficult to express in SQL [79]. This can be attributed to the use of a sophisticated bidirectional, sampling variant of Dijkstra's algorithm in Umbra's SNB BI implementation; this is formulated as a complex recursive SQL query (requiring 3,000 characters for Q15). Moreover, it is important to point out that while Umbra is a fully fledged RDBMS with high PostgreSQL compatibility, it is an academic prototype and lacks a mature storage system.

High level of scalability from TigerGraph. TigerGraph demonstrates a low memory usage: it is able to handle the SF1,000 data set with 1 TiB of total memory on 4 machines. Moreover, its loader and query engine can both scale out: using a 48-machine cluster, it is able to load the SF10,000 data set in less than 2 hours and complete the benchmark in less than 18 hours.

Read-heavy query mix. Table 4 shows that systems spend less than 5% of the execution time of the power batch on applying the updates (without precomputations). This implies that the workload's read–write ratio aligns with the general finding of survey [56] whose authors concluded that GDBMSs are typically used for read-intensive analytical queries. We also observed a difference between the systems: Umbra spends 0.8–1.6%, while TigerGraph spends 3.0–4.3% of the time on updates. This is due to a bottleneck in TigerGraph's execution of the deletes: while it processes the inserts distributedly, the input file of the delete operations is processed on a single machine, limiting the speed of cascading deletes.

6.5 Optimization Opportunities

We list some of the optimization opportunities observed during the implementation and execution of the benchmark. This is by no means a complete set of opportunities and we look forward to techniques proposed by users of this benchmark.

Precomputation. Both the Umbra and TigerGraph implementations utilized precomputations. To this end, the benchmark implementers created queries that materialize partial query results after applying the writes. As a general optimization, one precomputation eliminates tree traversals by calculating the explicit root Post of each Message during inserts. The implementations also used query-specific precomputations by calculating member counts per Country for each Forum (Q4), popularity scores for each Person (Q6), and the edge weights specified by Q19 and Q20. Additionally, TigerGraph precomputed the interaction scores defined in Q14 for each knows edge as part of its precomputation step for Q19.

Cheapest path algorithms. In SNB BI's cheapest path queries (Q15, Q19, Q20) path finding is executed from two (sets of) endpoints. Therefore, *bidirectional search algorithms* are beneficial. Q19 looks for paths between Persons of two Cities, therefore, it also benefits from multi-source *batched* path finding algorithms [65, 66]. Additionally, cheapest path queries can use *landmark labelling* [1], a method which precomputes cheapest paths for a set of *landmark* nodes, leading to improved pruning during runtime.

Factorization. Several queries, e.g. Q5, Q6, Q9, Q10, Q12, Q13, Q17, Q18, contain long paths which include many-to-many edges. These queries could benefit from applying *factorization* techniques [9, 58].

Multi-joins. Queries containing cyclic subgraphs along many-to-many edges (Q11, Q17, Q18) can benefit from *worst-case optimal multi-joins* [46]. Q18 is a particularly interesting candidate as it is among the top-5 most challenging query variants and could exploit both multi-joins and factorization.

6.6 Cost of Running the Experiments

Huppler argues that a good benchmark should be *economical* [30]. We strived to keep the cost of executing SNB BI at a reasonable level by optimizing the data generator and limiting the amount of time required for running the benchmark.

Table 4 shows an estimate of the costs for reproducing the experiments presented in this paper, assuming the pricing of the

AWS us-east-2 and GCP us-central1a regions as of October 2022. The total costs were derived based on the following items. *Storage cost:* AWS S3 costs $23.55/TiB/month, while GCP Cloud Storage is $26.0/TiB/month. *Execution cost:* for the Umbra experiments, the m6id.32xlarge instance costs $7.59/h, while for TigerGraph, each n2d-highmem-32 instance costs $1.82/h. We calculate the price of reserving these machines for the full duration of the benchmark plus account for a 1 hour setup time. Based on these, the estimated total cost of reproducing our experiments is $1,981.12.

Table 5: Key features tested and number of queries in related database benchmarks. *Notation:* ⊗ yes, ○ no, ⊘ limited coverage, ⊛ the benchmark provides a query generator. For cyclic subgraphs, only many-to-many edges are considered.

name	anti-/outer joins	cyclic subgraphs	complex paths	aggregation	global queries	correlated data	complex deletes	parameter curation	#read queries
LUBM [26]	○	○	○	○	○	○	○	○	14
SP^{2}Bench [59]	⊗	○	○	○	⊘	⊗	○	○	14
BSBM [10]	○	○	○	⊗	○	○	○	○	20
BSMA [76]	⊗	○	○	⊗	○	⊗	○	○	24
WatDiv [4]	⊗	○	○	⊗	⊘	⊘	○	○	⊛
TPC-H [68]	⊗	○	○	⊗	⊘	○	○	○	22
TPC-DS [50, 69]	⊗	○	○	⊗	⊘	○	○	○	99
LDBC SNB Interactive [20]	⊗	⊘	⊗	⊘	○	⊗	○	⊗	21
LDBC SNB BI [6, 64]	⊗	⊘	⊗	⊗	⊘	⊗	⊗	⊗	20

7 RELATED WORK

Transactional graph benchmarks. Several graph benchmarks define update-heavy transactional workloads: LinkBench [8], BG [2], and most recently TAOBench [13]. While most of these benchmarks use large-scale data sets (similarly to SNB BI), their workload only contains simple reads (lookups) and simple updates, hence, they are not representative of analytical graph processing workloads.

SPARQL and TPC analytical benchmarks. We summarize related SPARQL and analytical benchmarks in Table 5 and characterize them based on their coverage of graph-related and analytical features, as well as their support for correlated data generation, complex deletes, and parameter curation techniques. The table shows that none of the existing SPARQL benchmarks are representative of the challenges imposed by graph OLAP queries. Meanwhile, TPC's analytical benchmarks do not cover graph features, complex deletes or correlations, and rely on basic parameter generation algorithms.

Prior work on SNB BI. Some features and techniques in SNB BI, including the social network data model, some read queries, early-stage read-only experiments on data sets up to SF10, and the Datagen's approach to creating temporal graphs have been published in [6, 20, 25, 64, 75]. However, compared to [64], half of the queries have been replaced, primarily with ones stressing graph-specific choke points. Several other queries have been rewritten or improved, and the size of the largest experiment increased from SF10 to SF10,000. The update workload (including the cascading delete operations) and performance metrics are newly introduced.

8 CONCLUSION AND FUTURE WORK

We described the rationale for SNB BI, the first analytical graph benchmark that tests OLAP queries with graph tasks such as pattern matching and cheapest path finding, following sometimes edges that are generated on-the-fly. We presented highlights of its data generator, read queries, parameter curation techniques and recursive deletes. Additionally, we outlined its auditing process and presented preliminary experimental results. Our experiments included a large-scale distributed execution of SNB BI on SF10,000 using the TigerGraph GDBMS. Up to our best knowledge, this is the largest-scale *analytical GDBMS benchmark run* conducted to date. The experiments demonstrate that the benchmark is portable, scalable, and economical. The results show that running the benchmark is feasible for the current state-of-the-art, but our analysis suggests there is also headroom for improvement through research, e.g. incorporating factorization and fast worst-case optimal joins in graph-capable database systems.

Future work. An updated version of the SNB Interactive workload, which incorporates several of the innovations of SNB BI (large scale factors, delete operations, scalable parameter curation), is currently under development. Moreover, LDBC is designing the new *Financial Benchmark*, which targets distributed systems and runs financial fraud detection queries with real-time latency requirements [80].

ACKNOWLEDGMENTS

Gábor Szárnyas and Peter Boncz were partially supported by the SQIREL-GRAPHS NWO project. Jack Waudby was supported by the Engineering and Physical Sciences Research Council, Centre for Doctoral Training in Cloud Computing for Big Data [grant number EP/L015358/1]. We thank the following people for their participation: Renzo Angles, János Benjamin Antal, Alex Averbuch, Chaker Benhamad, Márton Búr, Thomas Cook, Hassan Chafi, Márton Elekes, Orri Erling, Michael Freitag, Gurbinder Gill, Andrey Gubichev, Vlad Haprian, Bálint Hegyi, Moritz Kaufmann, Josep Lluís Larriba-Pey, Norbert Martínez, József Marton, Amine Mhedhbi, Hannes Mühleisen, Thomas Neumann, Marcus Paradies, Arnau Prat-Pérez, Minh-Duc Pham, David Püroja, Mark Raasveldt, Oskar van Rest, Mirko Spasić, Vasileios Trigonakis, Danïel ten Wolde.

A QUERY SPECIFICATIONS

We give a concise summary of SNB BI's read query templates and update operations. Their detailed specification can be found in [6].

A.1 Read Queries

Q1: Posting summary. Find all Messages created before `$datetime`. Categorize them into 3 groups, (1) by year of creation, (2) by Message type, (3) by length (< 40, < 80, < 160, or longer).
Q2: Tag evolution. Find the Tags under a given `$tagClass` that were used in Messages during the 100-day time window starting at `$date` and the 100-day time window that follows. For the Tags and time windows, compute the count of Messages.

Q3: Popular topics in a country. Given a $tagClass and a $country, find all the Forums created in $country, containing at least one Message with Tags belonging to the given $tagClass, and count the Messages by their containing Forum.
Q4: Top message creators by country. Find the most popular Forums created after a given $date by Country, where the popularity of a Forum is measured by the number of members that Forum has in a given Country. Calculate the top-100 most popular Forums. For each member Person of the top-100 Forums, count the number of Messages they made in any of the top-100 Forums.
Q5: Most active posters of a given topic. Get each Person who created a Message with a given $tag. Considering only these Messages, for each Person, count (1) their Messages: mC, (2) likes to their Messages: lC, (3) reply Comments to their Messages: rC, then determine the score as $1 \cdot \texttt{mC} + 2 \cdot \texttt{rC} + 10 \cdot \texttt{lC}$.
Q6: Most authoritative users on a given topic. Given a $tag, find all Persons (person1) that ever created a Message with the $tag. For each person1 compute their "authority score" as follows: (1) The "authority score" is the sum of "popularity scores" of the Persons (person2) that liked any of that Person's Messages with $tag. (2) A Person's (person2) "popularity score" is defined as the total number of likes on their Messages.
Q7: Related topics. Find all Messages that have a given $tag. Find the related Tags attached to reply Comments of these Messages, but only consider ones that do not have the $tag. Group the Tags by name, and count the replies in each group.
Q8: Central person for a tag. Given a $tag, find all Persons who are interested in the $tag and/or have written a Message with a creationDate after a given $date with $tag. For each Person, compute the score as the sum of the following two aspects: (1) 100, if the Person has $tag among their interests, or 0 otherwise; (2) the number of Messages by the Person with $tag. For each Person, also compute the sum of the score of their friends.
Q9: Top thread initiators. For each Person, count their Posts and the number of Messages in each of their reply trees. For both Posts and Messages, filter on the time interval [$startDate, $endDate].
Q10: Experts in social circle. Given a $startPerson, find all other Persons (expert) who live in a given $country and are connected to $startPerson by a shortest path with length in range [$minPathDistance, $maxPathDistance] through the knows relation. For each of these expert nodes, retrieve all of their Messages that contain at least one Tag belonging to $tagClass. For each Message, retrieve all of its Tags. Count the Messages grouped by Person and Tag.
Q11: Friend triangles. See Section 3.2.1.
Q12: How many persons have a given number of messages. For each Person, count their Messages (mC) whose: (1) content is not empty, (2) length is below $lengthThreshold, (3) creationDate is after $startDate, (4) language is any of $languages. For each mC value, count the number of Persons with exactly mC Messages (with the required attributes).
Q13: Zombies in a country. Within $country, find zombies, i.e. Persons who were created before $endDate and created an average of $[0, 1)$ Messages per month during the time range between their creationDate and $endDate. For each zombie, calculate their zombieScore as the ratio between the number of likes received from other zombies and the total number of likes received.
Q14: International dialog. See Section 3.2.2.
Q15: Trusted connection paths through forums created in a given timeframe. Given $person1 and $person2, calculate the cost of the cheapest path between them along knows edges. The edges are weighted with $1/(\textit{interaction score} + 1)$, where the interaction score is calculated based on Message exchanges of the edge's Person endpoints: (1) every direct reply to a Post is 1.0 point and (2) every direct reply to a Comment is 0.5 points. Only consider Messages in Forums created during [$startDate, $endDate].
Q16: Fake news detection. Given ($tagA, $dateA) and ($tagB, $dateB), for both pairs ($tagX, $dateX), create an induced subgraph from the Person–knows–Person graph where both Persons have created a Message on the day of $dateX with $tagX. In the induced subgraph, only keep pairs of Persons who have at most $maxKnowsLimit friends. For these Persons, count the number of Messages created on $dateX with $tagX. Return Persons who had at least one Message for both ($tagA, $dateA) and ($tagB, $dateB).
Q17: Information propagation analysis. Find instances where person1 submitted a Message with a $tag to a Forum, and, at least $delta hours later, other members of forum1 engaged in a discussion (Message–Comment) with the same $tag in a different Forum where person1 is not a member. Return person1s with the number of discussions (potentially) triggered by them.
Q18: Friend recommendation. See Section 3.2.3.
Q19: Interaction path between cities. Given $city1 and $city2, find the person1 and person2 pairs living in these Cities with the cheapest interaction path between them. The path uses knows edges and the weight between two Persons is based on the rounded square root of the number of interactions (reply Comments to a Message by the other Person). At least one interaction is required.
Q20: Recruitment. See Section 3.2.4.

A.2 Insert Operations

In each batch, the insert operations insert entities created during one day of simulation time (Person nodes, knows edges, etc.). In *concurrent RW mode*, systems must perform the insertions such that the graph is well-formed after each operation (e.g. there are no dangling Forums and Messages). In *disjoint RW mode*, systems are free to perform these operations in any order.

A.3 Delete Operations

DEL1: Remove a Person and its edges. Additionally, remove the Album/Wall Forums whose moderator is the Person and remove all Messages the Person has created in the rest of the Forums.
DEL2: Given a Person and a Post, remove their likes edge.
DEL3: Given a Person and a Comment, remove their likes edge.
DEL4: Remove a Forum, its edges, and all Posts in the Forum, and all Comments in their threads.
DEL5: Given a Forum and a Person, remove their hasMember edge.
DEL6: Remove a Post node and its edges (isLocatedIn, likes, hasCreator, hasTag, containerOf). Remove all (direct and transitive) reply Comments to the Post and their edges.
DEL7: Remove a Comment node, its edges (isLocatedIn, likes, hasCreator, hasTag), and all replies to the Comment.
DEL8: Given two Person nodes, remove their knows edge.

REFERENCES

[1] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*. ACM, 349–360. https://doi.org/10.1145/2463676.2465315

[2] Yazeed Alabdulkarim, Sumita Barahmand, and Shahram Ghandeharizadeh. 2018. BG: A scalable benchmark for interactive social networking actions. *Future Gener. Comput. Syst.* 85 (2018), 29–38. https://doi.org/10.1016/j.future.2018.02.031

[3] Hazim Almuhimedi, Shomir Wilson, Bin Liu, Norman M. Sadeh, and Alessandro Acquisti. 2013. Tweets are forever: A large-scale quantitative analysis of deleted tweets. In *CSCW*. ACM, 897–908. https://doi.org/10.1145/2441776.2441878

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

[5] Renzo Angles. 2018. The Property Graph Database Model. In *AMW (CEUR Workshop Proceedings)*, Vol. 2100. CEUR-WS.org. http://ceur-ws.org/Vol-2100/paper26.pdf

[6] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba-Pey, Norbert Martínez-Bazan, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasic, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. 2020. The LDBC Social Network Benchmark. *CoRR* abs/2001.02299 (2020). http://arxiv.org/abs/2001.02299

[7] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. 2018. G-CORE: A Core for Future Graph Query Languages. In *SIGMOD*. ACM, 1421–1432. https://doi.org/10.1145/3183713.3190654

[8] 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

[9] Nurzhan Bakibayev, Dan Olteanu, and Jakub Zavodny. 2012. FDB: A Query Engine for Factorised Relational Databases. *Proc. VLDB Endow.* 5, 11 (2012), 1232–1243. https://doi.org/10.14778/2350229.2350242

[10] Christian Bizer and Andreas Schultz. 2009. The Berlin SPARQL Benchmark. *Int. J. Semantic Web Inf. Syst.* 5, 2 (2009), 1–24. https://doi.org/10.4018/jswis.2009040101

[11] Peter A. Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark, Vol. 8391. Springer, 61–76. https://doi.org/10.1007/978-3-319-04936-6_5

[12] Federico Busato, Oded Green, Nicola Bombieri, and David A. Bader. 2018. Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs. In *HPEC*. IEEE, 1–7. https://doi.org/10.1109/HPEC.2018.8547541

[13] Audrey Cheng et al. 2022. TAOBench: An End-to-End Benchmark for Social Networking Workloads". In *VLDB*.

[14] Debezium Community. 2022. Debezium. https://debezium.io/

[15] 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

[16] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *SIGMOD*. ACM, 2246–2258. https://doi.org/10.1145/3514221.3526057

[17] 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

[18] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2020. Aggregation Support for Modern Graph Analytics in TigerGraph. In *SIGMOD*. ACM, 377–392. https://doi.org/10.1145/3318464.3386144

[19] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *VLDB* 13, 8 (2020), 1206–1220. http://www.vldb.org/pvldb/vol13/p1206-dreseler.pdf

[20] Orri Erling, Alex Averbuch, Josep-Lluis 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*. 619–630. https://doi.org/10.1145/2723372.2742786

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

[22] Jim Gray (Ed.). 1993. *The Benchmark Handbook for Database and Transaction Systems* (2nd ed.). Morgan Kaufmann.

[23] Jim Gray. 2005. A "Measure of Transaction Processing" 20 Years Later. *IEEE Data Eng. Bull.* 28, 2 (2005), 3–4. http://sites.computer.org/debull/A05june/gray.ps

[24] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals. *Data Min. Knowl. Discov.* 1, 1 (1997), 29–53. https://doi.org/10.1023/A:1009726021843

[25] Andrey Gubichev and Peter A. Boncz. 2014. Parameter Curation for Benchmark Queries. In *TPCTC (Lecture Notes in Computer Science)*, Vol. 8904. Springer, 113–129. https://doi.org/10.1007/978-3-319-15350-6_8

[26] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.* 3, 2-3 (2005), 158–182. https://doi.org/10.1016/j.websem.2005.06.005

[27] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. 2013. WTF: The who to follow service at Twitter. In *WWW*. International World Wide Web Conferences Steering Committee / ACM, 505–514. https://doi.org/10.1145/2488388.2488433

[28] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. 2021. Columnar Storage and List-based Processing for Graph Database Management Systems. *Proc. VLDB Endow.* 14, 11 (2021), 2491–2504. https://doi.org/10.14778/3476249.3476297

[29] Rihan Hai, Christoph Quix, and Matthias Jarke. 2021. Data lake concept and systems: A survey. *CoRR* abs/2106.09592 (2021). https://arxiv.org/abs/2106.09592

[30] Karl Huppler. 2009. The Art of Building a Good Benchmark. In *TPCTC*. 18–30. https://doi.org/10.1007/978-3-642-10424-4_3

[31] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardt, Hassan Chafi, Mihai Capota, Narayanan Sundaram, Michael J. Anderson, Ilie Gabriel Tanase, Yinglong Xia, Lifeng Nai, and Peter A. Boncz. 2016. LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. *PVLDB* 9, 13 (2016), 1317–1328. https://doi.org/10.14778/3007263.3007270

[32] Alexandru Iosup, Ahmed Musaafir, Alexandru Uta, Arnau Prat-Pérez, Gábor Szárnyas, Hassan Chafi, Ilie Gabriel Tanase, Lifeng Nai, Michael J. Anderson, Mihai Capota, Narayanan Sundaram, Peter A. Boncz, Siegfried Depner, Stijn Heldens, Thomas Manhardt, Tim Hegeman, Wing Lung Ngai, and Yinglong Xia. 2020. The LDBC Graphalytics Benchmark. *CoRR* abs/2011.15028 (2020). https://arxiv.org/abs/2011.15028

[33] Jeremy Kepner, Peter Aaltonen, David A. Bader, Aydin Buluç, Franz Franchetti, John R. Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Carl Yang, John D. Owens, Marcin Zalewski, Timothy G. Mattson, and José E. Moreira. 2016. Mathematical foundations of the GraphBLAS. In *HPEC*. IEEE. https://doi.org/10.1109/HPEC.2016.7761646

[34] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: Fast compilation and fast execution of relational queries in Umbra. *VLDB J.* 30, 5 (2021), 883–905. https://doi.org/10.1007/s00778-020-00643-4

[35] LDBC. 2021. Byelaws of the Linked Data Benchmark Council v1.3. https://ldbcouncil.org/docs/LDBC.Byelaws.1.3.ADOPTED.2021-01-14.pdf

[36] Jure Leskovec, Lars Backstrom, and Jon M. Kleinberg. 2009. Meme-tracking and the dynamics of the news cycle. In *SIGKDD*. ACM, 497–506. https://doi.org/10.1145/1557019.1557077

[37] László Lőrincz, Júlia Koltai, Anna Fruzsina Győr, and Károly Takács. 2019. Collapse of an online social network: Burning social capital to create it? *Soc. Networks* 57 (2019), 43–53. https://doi.org/10.1016/j.socnet.2018.11.004

[38] M. McPherson, L. Smith-Lovin, and J. M. Cook. 2001. Birds of a feather: Homophily in social networks. *Annual Review of Sociology* (2001), 415–444.

[39] 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

[40] David Mizell, Kristyn J. Maschhoff, and Steven P. Reinhardt. 2014. Extending SPARQL with graph functions. In *BigData*. IEEE Computer Society, 46–53. https://doi.org/10.1109/BigData.2014.7004371

[41] Seth A. Myers and Jure Leskovec. 2014. The bursty dynamics of the Twitter information network. In *WWW*. ACM, 913–924. https://doi.org/10.1145/2566486.2568043

[42] Thomas Neumann. 2021. Evolution of a Compiling Query Engine. *Proc. VLDB Endow.* 14, 12 (2021), 3207–3210. https://doi.org/10.14778/3476311.3476410

[43] 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

[44] Thomas Neumann and Alfons Kemper. 2015. Unnesting Arbitrary Queries. In *BTW (LNI)*, Vol. P-241. GI, 383–402. https://dl.gi.de/20.500.12116/2418

[45] Thomas Neumann and Bernhard Radke. 2018. Adaptive Optimization of Very Large Join Queries. In *SIGMOD*. ACM, 677–692. https://doi.org/10.1145/3183713.3183733

[46] 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

[47] Dan Olteanu and Maximilian Schleich. 2016. Factorized Databases. *SIGMOD Rec.* 45, 2 (2016), 5–16. https://doi.org/10.1145/3003665.3003667

[48] Oracle. 2022. GoldenGate. https://www.oracle.com/integration/goldengate/

[49] Minh-Duc Pham, Peter A. Boncz, and Orri Erling. 2012. S3G2: A Scalable Structure-Correlated Social Graph Generator. In *TPCTC*. 156–172. https://doi.org/10.1007/978-3-642-36727-4_11

[50] Meikel Pöss, Tilmann Rabl, and Hans-Arno Jacobsen. 2017. Analysis of TPC-DS: the first standard benchmark for SQL-based big data systems. In *SoCC*. 573–585. https://doi.org/10.1145/3127479.3128603

[51] Mark Raasveldt and Hannes Mühleisen. 2017. Don't Hold My Data Hostage - A Case For Client Protocol Redesign. *Proc. VLDB Endow.* 10, 10 (2017), 1022–1033. https://doi.org/10.14778/3115404.3115408

[52] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *SIGMOD*. ACM, 1981–1984. https://doi.org/10.1145/3299869.3320212

[53] Liam Roditty and Uri Zwick. 2004. On Dynamic Shortest Paths Problems. In *ESA (Lecture Notes in Computer Science)*, Vol. 3221. Springer, 580–591. https://doi.org/10.1007/978-3-540-30140-0_52

[54] Marko A. Rodriguez. 2015. The Gremlin graph traversal machine and language (invited talk). In *DBPL*. 1–10. https://doi.org/10.1145/2815072.2815073

[55] Yousef Saad. 2003. *Iterative methods for sparse linear systems*. SIAM. https://doi.org/10.1137/1.9780898718003

[56] 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, 2-3 (2020), 595–618. https://doi.org/10.1007/s00778-019-00548-x

[57] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid G. Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. 2021. The future is big graphs: A community view on graph processing systems. *Commun. ACM* 64, 9 (2021), 62–71. https://doi.org/10.1145/3434642

[58] Maximilian Schleich and Dan Olteanu. 2020. LMFAO: An Engine for Batches of Group-By Aggregates. *Proc. VLDB Endow.* 13, 12 (2020), 2945–2948. https://doi.org/10.14778/3415478.3415515

[59] Michael Schmidt, Thomas Hornung, Michael Meier, Christoph Pinkel, and Georg Lausen. 2009. SP^2Bench: A SPARQL Performance Benchmark. In *Semantic Web Information Management - A Model-Based Perspective*. Springer, 371–393. https://doi.org/10.1007/978-3-642-04329-1_16

[60] Juan Sequeda and Ora Lassila. 2021. *Designing and Building Enterprise Knowledge Graphs*. Morgan & Claypool Publishers. https://doi.org/10.2200/S01105ED1V01Y202105DSK020

[61] Supreeth Shastri, Vinay Banakar, Melissa Wasserman, Arun Kumar, and Vijay Chidambaram. 2020. Understanding and Benchmarking the Impact of GDPR on Database Systems. *VLDB* 13, 7 (2020), 1064–1077. http://www.vldb.org/pvldb/vol13/p1064-shastri.pdf

[62] Dávid Szakállas. 2020. Speeding Up LDBC SNB Datagen. https://ldbcouncil.org/post/speeding-up-ldbc-snb-datagen/

[63] Dávid Szakállas. 2022. LDBC SNB Datagen – The Winding Path to SF100k. https://ldbcouncil.org/post/ldbc-snb-datagen-the-winding-path-to-sf100k/

[64] Gábor Szárnyas, Arnau Prat-Pérez, Alex Averbuch, József Marton, Marcus Paradies, Moritz Kaufmann, Orri Erling, Peter A. Boncz, Vlad Haprian, and János Benjamin Antal. 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

[65] Manuel Then, Stephan Günnemann, Alfons Kemper, and Thomas Neumann. 2017. Efficient Batched Distance and Centrality Computation in Unweighted and Weighted Graphs. In *BTW (LNI)*, Vol. P-265. GI, 247–266. https://dl.gi.de/20.500.12116/632

[66] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T. Vo. 2014. The More the Merrier: Efficient Multi-Source Graph Traversal. *Proc. VLDB Endow.* 8, 4 (2014), 449–460. https://doi.org/10.14778/2735496.2735507

[67] TPC (Transaction Processing Performance Council). 2010. TPC Benchmark C, revision 5.11. , 132 pages. https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf

[68] TPC (Transaction Processing Performance Council). 2017. TPC Benchmark H, revision 2.18.0. (2017), 1–138. https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.18.0.pdf

[69] TPC (Transaction Processing Performance Council). 2019. TPC Benchmark DS, revision 2.11.0. (2019), 1–141. https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v2.11.0.pdf

[70] TPC (Transaction Processing Performance Council). 2021. TPC Pricing Specification, revision 2.7.0. (2021), 1–62. http://tpc.org/TPC_Documents_Current_Versions/pdf/TPC-Pricing_v2.7.0.pdf

[71] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: A property graph query language. In *GRADES at SIGMOD*. https://doi.org/10.1145/2960414.2960421

[72] Todd L. Veldhuizen. 2014. Leapfrog Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *ICDT*. OpenProceedings.org, 96–106. https://doi.org/10.5441/002/icdt.2014.13

[73] D. J. Watts and S. H. Strogatz. 1998. Collective dynamics of 'small-world' networks. *Nature* 393 (1998), 440–442.

[74] Jack Waudby, Benjamin A. Steer, Karim Karimov, József Marton, Peter A. Boncz, and Gábor Szárnyas. 2020. Towards Testing ACID Compliance in the LDBC Social Network Benchmark. In *TPCTC*. Springer, 1–17. https://doi.org/10.1007/978-3-030-84924-5_1

[75] 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

[76] Fan Xia, Ye Li, Chengcheng Yu, Haixin Ma, and Weining Qian. 2014. BSMA: A Benchmark for Analytical Queries over Social Media Data. *Proc. VLDB Endow.* 7, 13 (2014), 1573–1576. https://doi.org/10.14778/2733004.2733033

[77] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. 2013. GraphX: A resilient distributed graph system on Spark. In *GRADES at SIGMOD*. CWI/ACM. https://doi.org/10.1145/2484425.2484427

[78] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65. https://doi.org/10.1145/2934664

[79] Kangfei Zhao and Jeffrey Xu Yu. 2017. Graph Processing in RDBMSs. *IEEE Data Eng. Bull.* 40, 3 (2017), 6–17. http://sites.computer.org/debull/A17sept/p6.pdf

[80] Xiaowei Zhu, Zhisong Fu, Zhenxuan Pan, Jin Jiang, Chuntao Hong, Yongchao Liu, Yang Fang, Wenguang Chen, and Changhua He. 2021. Taking the Pulse of Financial Activities with Online Graph Processing. *ACM SIGOPS Oper. Syst. Rev.* 55, 1 (2021), 84–87. https://doi.org/10.1145/3469379.3469389