The LDBC Social Network Benchmark: Interactive Workload

Orri Erling
OpenLink Software, UK
oerling@openlinksw.com

Alex Averbuch
Neo Technology, Sweden
alex.averbuch@neotechnology.com

Josep Larriba-Pey
Sparsity Technologies, Spain
larri@sparsity-technologies.com

Hassan Chafi
Oracle Labs, USA
hassan.chafi@oracle.com

Andrey Gubichev
TU Munich, Germany
gubichev@in.tum.de

Arnau Prat*
Universitat Politècnica de Catalunya, Spain
aprat@ac.upc.edu

Minh-Duc Pham
VU University Amsterdam, The Netherlands
m.d.pham@vu.nl

Peter Boncz
CWI, Amsterdam, The Netherlands
boncz@cwi.nl

ABSTRACT

The Linked Data Benchmark Council (LDBC) is now two years underway and has gathered strong industrial participation for its mission to establish benchmarks, and benchmarking practices for evaluating graph data management systems. The LDBC introduced a new *choke-point* driven methodology for developing benchmark workloads, which combines user input with input from expert systems architects, which we outline. This paper describes the LDBC Social Network Benchmark (SNB), and presents database benchmarking innovation in terms of graph query functionality tested, correlated graph generation techniques, as well as a scalable benchmark driver on a workload with complex graph dependencies. SNB has three query workloads under development: Interactive, Business Intelligence, and Graph Algorithms. We describe the SNB Interactive Workload in detail and illustrate the workload with some early results, as well as the goals for the two other workloads.

1. INTRODUCTION

Managing and analyzing graph-shaped data is an increasingly important use case for many organizations, in for instance marketing, fraud detection, logistics, pharma, healthcare but also digital forensics and security. People have been trying to use existing technologies, such as relational database systems for graph data management problems. It is perfectly possible to represent and store a graph in a relational table, for instance as a table where every row contains an edge, and the start and end vertex of every edge are a foreign key reference (in SQL terms). However, what makes a data management problem a graph problem is that the data analysis is not only about the values of the data items in such a table, but about the *connection patterns* between the various pieces. SQL-based systems were not originally designed for this – though systems have implemented diverse extensions for navigational and recursive query execution.

In recent years, the database industry has seen a proliferation of new graph-oriented data management technologies. Roughly speaking, there are four families of approaches. One are pure graph database systems, such as Neo4j, Sparksee and Titan, which elevate graphs to first class citizens in their data model ("property graphs"), query languages, and APIs. These systems often provide specific features such as breadth-first search and shortest path algorithms, but also allow to insert, delete and modify data using transactional semantics. A second variant are systems intended to manage semantic web data conforming to the RDF data model, such as Virtuoso or OWLIM. Although RDF systems emphasize usage in semantic applications (e.g. data integration), RDF is a graph data model, which makes SPARQL the only well-defined standard query language for graph data. A third kind of new system targets the need to compute certain complex graph algorithms, that are normally not expressed in high-level query languages, such as Community Finding, Clustering and PageRank, on huge graphs that may not fit the memory of a single machine, by making use of cluster computing. Example systems are GraphLab, Stratosphere and Giraph, though this area is still heavily in motion and does not yet have much industrial installed base. Finally, recursive SQL, albeit not very elegant, is expressive enough to construct a large class of graph queries (variable length path queries, pattern matching, etc.). One of the possibilities (exemplified by Virtuoso RDBMS) is to introduce vendor-specific extensions to SQL, which are basically shortcuts for recursive SQL subqueries to run specific graph algorithms inside SQL queries (such as shortest paths).

*Supported by Oracle Labs

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

http://dx.doi.org/10.1145/2723372.2742786.

The Linked Data Benchmark Council[1] (LDBC) is an independent authority responsible for specifying benchmarks, benchmarking procedures and verifying/publishing benchmark results. Benchmarks on the one hand allow to quantitatively compare different technological solutions, helping IT users to make more objective choices for their software architectures. On the other hand, an important second goal for LDBC is to stimulate technological progress among competing systems and thereby accelerate the maturing of the new software market of graph data management systems.

This paper describes the Social Network Benchmark (SNB), the first LDBC benchmark, which models a social network akin to Facebook. The dataset consists of persons and a *friendship network* that connects them; whereas the majority of the data is in the *messages* that these persons post in discussion trees on their forums. While SNB goes through lengths to make its generated data more realistic than previous synthetic approaches, it should not be understood as an attempt to fully model Facebook – its ambition is to be as realistic as necessary for the benchmark queries to exhibit the desired effects – nor does the choice for social network data as the scenario for SNB imply that LDBC sees social network companies as the primary consumers of its benchmarks – typically these internet-scale companies do not work with standard data management software and rather roll their own. Rather, the SNB scenario is chosen because it is an appealing graph-centric use case, and in fact social network analysis on data that contains excerpts of social networks is a very common marketing activity nowadays.

There are in fact three SNB benchmarks on one common dataset, since SNB has three different *workloads*. Each workload produces a single metric for performance at the given scale and a price/performance metric at the scale and can be considered a separate benchmark. The full disclosure further breaks down the composition of the metric into its constituent parts, e.g. single query execution times.

SNB-Interactive. This workload consists of a set of relatively complex read-only queries, that touch a significant amount of data, often the two-step friendship neighborhood and associated messages. Still these queries typically start at a single point and the query complexity is sublinear to the dataset size. Associated with the complex read-only queries are simple read-only queries, which typically only lookup one entity (e.g. a person). Concurrent with these read-only queries is an insert workload, under at least read committed transaction semantics. All data generated by the SNB data generator is timestamped, and a standard scale factor covers three years. Of this 32 months are bulkloaded at benchmark start, whereas the data from the last 4 months is added using individual DML statements.

SNB-BI. This workload consists of a set of queries that access a large percentage of all entities in the dataset (the "fact tables"), and groups these in various dimensions. In this sense, the workload has similarities with existing relational Business Intelligence benchmarks like TPC-H and TPC-DS; the distinguishing factor is the presence of graph traversal predicates and recursion. Whereas the SNB Interactive workload has been fully developed, the SNB BI workload is a working draft, and the concurrent bulk-load workload has not yet been specified.

[1]ldbcouncil.org - LDBC originates from the EU FP7 project (FP7-317548) by the same name.

SNB-Algorithms. This workload is under construction, but is planned to consist of a handful of often-used graph analysis algorithms, including PageRank, Community Detection, Clustering and Breadth First Search. While we foresee that the two other SNB workloads can be used to compare graph database systems, RDF stores, but also SQL stores or even noSQL systems; the SNB-Algorithms workload primary targets graph programming systems or even general purpose cluster computing environments like MapReduce. It may, however, be possible to implement graph algorithms as iterative queries, e.g. keeping state in temporary tables, hence it is possible that other kinds of systems may also implement it.

Given that graph queries and graph algorithm complexity is heavily influenced by the complex structure of the graph, we specifically aim to run all three benchmarks on the same dataset. In the process of benchmark definition, the dataset generator is being tuned such that the graph, e.g. contains communities, and clusters comparable to clusters and communities found on real data. These graph properties cause the SNB-Algorithms workload to produce "sensible" results, but are also likely to affect the behavior of queries in SNB-Interactive and SNB-BI. Similarly, the graph degree and value/structure correlation (e.g. people having names typical for a country) that affect query outcomes in SNB-Interactive and BI may also implicitly affect the complexity of SNB-Algorithms. As such, having three diverse workloads on the same dataset is thought to make the behavior of all workloads more realistic, even if we currently would not understand or foresee how complex graph patterns affect all graph management tasks.

This paper focuses on SNB-Interactive, since this workload is complete. The goal of SNB-Interactive is to test graph data management systems that combine transactional update with query capabilities. A well-known graph database system that offers this is Neo4j, but SNB-Interactive is formulated such that many systems can participate, as long a they support transactional updates allowing simultaneous queries. The query workload focus on interactivity, with the intention of sub-second response times and query patterns that start typically at a single graph node and visit only a small portion of the entire graph. One could hence position it as OLTP, even though the query complexity is much higher than TPC-C and does include graph tasks such as traversals and restricted shortest paths. The rationale for this focus stems from LDBC research among its vendor members and the LDBC Technical User Community of database users. This identified that many interactive graph applications currently rely on key-value data management systems without strong consistency, where query predicates that are more complex than a key-lookup are answered using offline pre-computed data. This staleness and lack of consistency both impact the user experience and complicate application development, hence LDBC hopes that SNB-Interactive will lead to the maturing of transactional graph data management systems that can improve the user experience and ease application development.

The main contributions by the LDBC work on SNB are the following:

scalable correlated graph. The SNB graph generator has been shown to be much more realistic than previous synthetic data generators [13], for which reason it was already chosen to be the base of the 2014 SIGMOD programming

contest. The graph generator is further notable because it realizes well-known power laws, uses skewed value distributions, but also introduces plausible *correlations* between property values and graph structures.

choke-point based design. The SNB-Interactive query workload has been carefully designed according to so-called *choke-point* analysis that identifies important technical challenges to evaluate in a workload. This analysis requires both user input[2] as well as expert input from database systems architects. In defining the SNB-Interactive, LDBC has worked with the core architects of Neo4j, RDF-3X, Virtuoso, Sparksee, MonetDB, Vectorwise and HyPer.

dependency synchronization. The SNB query driver solves the difficult task of generating a highly parallel workload to achieve high throughput, on a datasets that by its complex connected component structure is impossible to partition. This could easily lead to extreme overhead in the query driver due to synchronization between concurrent client threads and processes – the SNB driver enables optimizations that strongly reduce the need for such synchronization by identifying *sequential* and *window*-based execution modes for parts of the workload.

parameter curation. Since the SNB dataset is such a complex graph, with value/structure correlations affecting queries over the friends graph and message discussion trees, with most distributions being either skewed (typically using the exponential distribution) or power-laws, finding good query parameters is non-trivial. If uniformly chosen values would serve as parameters, the complexity of any query template would vary enormously between the parameters – an undesirable phenomenon for the understandability of a benchmark. The SNB therefore introduced a new benchmarking concept, namely *Parameter Curation* [6] that performs a data mining step during data generation to find substitution parameters with equivalent behavior.

2. INNOVATIVE GRAPH GENERATOR

The LDBC SNB data generator (DATAGEN) evolved from the S3G2 generator [10] and simulates the user's activity in a social network during a period of time. Its schema has 11 entities connected by 20 relations, with attributes of different types and values, making for a rich benchmark dataset. The main entities are: Persons, Tags, Forums, Messages (Posts, Comments and Photos), Likes, Organizations, and Places. A detailed description of the schema is found at [11].

The dataset forms a graph that is a *fully connected component* of persons over their *friendship relationships*. Each person has a few forums under which the messages form large discussion *trees*. The messages are further connected to posts by authorship but also likes. These data elements scale linearly with the amount of friendships (people having more friends are likely more active and post more messages). Organization and Place information are more dimension-like and do not scale with the amount of persons or time. Time is an implicit dimension (there is no separate time entity) but is present in many timestamp attributes.

[2]LDBC has a Technical User Community which it consults for input and feedback.

(person.location, person.gender)	person.firstName	*(typical names)*
	person.interests	*(popular artist)*
person.location	person.lastName	*(typical names)*
	person.university	*(nearby universities)*
	person.company	*(in country)*
	person.languages	*(spoken in country)*
person.language	person.forum.post.language	*(speaks)*
person.interests	person.forum.post.topic	*(in)*
post.topic	post.text	*(DBpedia article lines)*
	post.comment.text	*(DBpedia article lines)*
person.employer	person.email	*(@company, @university)*
post.photoLocation	post.location.latitude	*(matches location)*
	post.location.longitude	*(matches location)*
person.birthDate	person.createdDate	*(>)*
person.createdDate	person.forum.message.createdDate	*(>)*
	person.forum.createdDate	*(>)*
forum.createdDate	post.photoTime	*(>)*
	forum.post.createdDate	*(>)*
	forum.groupmembership.joinedDate	*(>)*
post.createdDate	post.comment.createdDate	*(>)*

Table 1: Attribute Value Correlations: left determines right

Name	Number
Karl	215
Hans	190
Wolfgang	174
Fritz	159
Rudolf	159
Walter	150
Franz	115
Paul	109
Otto	99
Wilhelm	74

Name	Number
Yang	961
Chen	929
Wei	887
Lei	789
Jun	779
Jie	778
Li	562
Hao	533
Lin	456
Peng	448

Table 2: Top-10 person.firstNames (SF=10) for persons with person.location=Germany (left) or China (right).

2.1 Correlated Attribute Values

An important novelty in DATAGEN is the ability to produce a highly correlated social network graph, in which attribute values are correlated among themselves and also influence the connection patterns in the social graph. Such correlations clearly occur in real graphs and influence the complexity of algorithms operating on the graph.

A full list of attribute correlations is given in Table 1. For instance, the top row in the table states that the place where a person was born and gender influence the first name distribution. An example is shown in Table 2, which shows the top-10 most occurring first names for people from Germany vs China. The actual set of attribute values is taken from DBpedia, which also is used as a source for many other attributes. Similarly, the location where a person lives influences his/her interests (a set of tags), which in turn influences the topic of the discussions (s)he opens (i.e., Posts), which finally also influences the text of the messages in the discussion. This is implemented by using the text taken from DBpedia pages closely related to a topic as the text used in the discussion (original post and comments on it).

Person location also influences last name, university, company and languages. This influence is not full, there are Germans with Chinese names, but these are infrequent. In fact, the shape of the attribute value distributions is equal (and skewed), but the order of the values from the *value dictionaries* used in the distribution, changes depending on the correlation parameters (e.g. location).

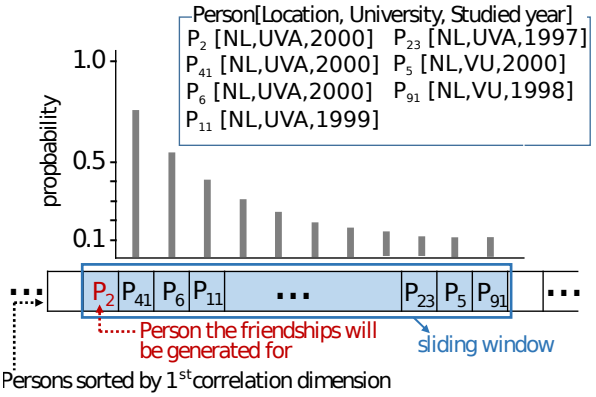


Figure 1: Friendships generation (NL: The Netherlands, UVA: University of Amsterdam, VU: Vrij University)

2.2 Time Correlation and Spiking Trends

Almost all entities in the SNB dataset have timestamp attributes, since time is an important phenomenon in social networks. The latter correlation rules in Table 1 are related to time, and ensure that events in the social network follow a logical order: e.g., people can post a comment only after becoming a friend with someone, and that can only happen after both persons joined the network.

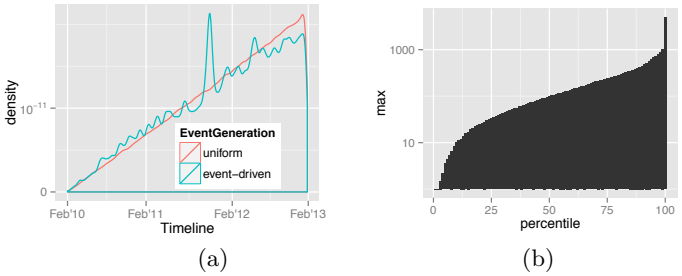
The volume of person activity in a real social network, i.e., number of messages created per unit of time, is not uniform, but driven by real world events such as elections, natural disasters and sport competitions. Whenever an important real world event occurs, the amount of people and messages talking about that topic spikes – especially from those persons interested in that topic. We introduced this in DATAGEN by simulating events related to certain tags, around which the frequency of posts by persons interested in that tag is significantly higher (the topic is "trending"). Figure 2(a) shows the density of posts over time with and without event-driven post generation, for SF=10. When event driven post generation is enabled, the density is not uniform but spikes of different magnitude appear, which correspond to events of different levels of importance. The activity volume around an event is implemented as proposed in [7].

2.3 Structure Correlation: Friendships

The "Homophily Principle" [8] states that similar people have a higher probability to be connected. This is modeled by DATAGEN by making the probability that people are connected dependent on their characteristics (attributes). This is implemented by a multi-stage edge generation process over two **correlation dimensions**: (i) *places where people studied* and (ii) *interests of persons*.

In other words, people that are interested in a topic and/or have studied in the same university at the same year, have a larger probability to be friends. Furthermore, in order to reproduce the inhomogeneities found in real data, a third dimension consisting of a random number is also used.

In each edge generation stage the persons are re-sorted on one dimension (first stage: study location, second: interests, last: random). Each worker processes a disjunct range of these persons sequentially, keeping a window of the persons in memory – the entire range does not have to fit – and picks friends from the window using a geometric probability distribution that decreases with distance in the window.



(a) (b)

Figure 2: (a) Post distribution over time for event-driven vs uniform post generation on SF=10. (b) Maximum degree of each percentile in the Facebook graph.

The probability for generating a connection during this stage drops from very low at window boundary to zero outside it (since the generator is not even capable of generating a friendship to data dropped from its window). All this makes the complex task of generating correlated friendship edges scalable, as it now only depends on parallel sorting and sequential processing with limited memory. We note that one dimension may have the form of multiple single-dimensional values bitwise appended. In the particular case of the *studied location*, these are the Z-order location of the university's city (bits 31-24), the university ID (bits 23-12), and the studied year (bits 11-0). This is exemplified at Figure 1 where we show a sliding window along the first correlation dimension (i.e., *studied location*). As shown in this figure, those persons closer to person P_2 (the person generating friends for) according to the first dimension (e.g., P_{41}, P_6) have a higher probability to be friends of P_2.

The correlations in the friends graph also propagate to the messages. A person location influences on the one hand interests and studied location, so one gets many more like-minded or local friends. These persons typically have many more common interests (tags), which become the topic of posts and comment messages.

The *number* of friendship edges generated per person (friendship degree) is skewed [4]. DATAGEN discretizes the power law distribution given by Facebook graph [14], but scales this according to the size of the network. Because in smaller networks, the amount of "real" friends that is a member and to which one can connect is lower, we adjust the mean average degree logarithmically in terms of person membership, such that it becomes (somewhat) lower for smaller networks. A target average degree of the friendship graph is chosen using the following formula: $avg\_degree = n^{0.512-0.028 \cdot log(n)}$, where n is the number of persons in the graph. That is, when the size of the SNB dataset would be that of Facebook (i.e. 700M persons) the average friendship degree would be around 200. Then, each person is first assigned to a percentile p in the Facebook's degree distribution and second, a target degree uniformly distributed between the minimum and the maximum degrees at percentile p. Figure 2(b) shows the maximum degree per percentile of the Facebook graph, used in DATAGEN. Finally, the person's target degree is scaled by multiplying it by a factor resulting from dividing *avg_degree* by the average degree of the real Facebook graph. Figure 3(a) shows the friendship degree distribution for SF=10. Finally, given a person, the number of friendship edges for each correlation dimension is distributed as follows: 45%, 45% and 10% out of the target degree, for the first, the second and the third correlation dimension, respectively.

SFs	Number of entities (x 1000000)					
	Nodes	Edges	Persons	Friends	Messages	Forums
30	99.4	655.4	0.18	14.2	97.4	1.8
100	317.7	2154.9	0.50	46.6	312.1	5.0
300	907.6	6292.5	1.25	136.2	893.7	12.6
1000	2930.7	20704.6	3.60	447.2	2890.9	36.1

Table 3: SNB dataset statistics at different Scale Factors

2.4 Scales & Scaling

DATAGEN can generate social networks of arbitrary size, however for the benchmarks we work with standard scale-factors (SF) valued 1,3,10,30,.. as indicated in Table 3. The scale is determined by setting the amount of persons in the network, yet the scale factor is the amount of GB of uncompressed data in comma separated value (CSV) representation. DATAGEN can also generate RDF data in Ntriple[3] format, which is much more verbose.

DATAGEN is implemented on top of Hadoop to provide scalability. Data generation is performed in three steps, each of them composed of more MapReduce jobs.

person generation: In this step, the people of the social network are generated, including the personal information, interests, universities where they studied and companies where they worked at. Each mapper is responsible of generating a subset of the persons of the network.

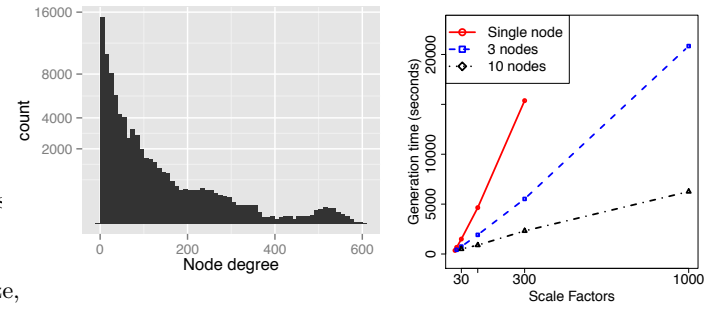
friendship generation: As explained above, friendship generation is split into a succession of stages, each of them based on a different correlation dimension. Each of these stages consists of two MapReduce jobs. The first is responsible for sorting the persons by the given correlation dimension. The second receives the sorted people and performs the sliding window process explained above.

person activity generation: this involves filling the forums with posts comments and likes. This data is mostly tree-structured and is therefore easily parallelized by the person who owns the forum. Each worker needs the attributes of the owner (e.g. interests influence post topics), the friend list (only friends post comments and likes) with the friendship creation timestamps (they only post after that); but otherwise the workers can operate independently.

We have paid specific attention to making data generation *deterministic*. This means that regardless the Hadoop configuration parameters (#node, #map and #reduce tasks) the generated dataset is always the same.

On a single 4-core machine (Intel i7-2600K@3.4GHz, 16GB RAM) that runs MapReduce in "pseudo-distributed" mode – where each CPU core runs a mapper or reducer – one can generate a SF=30 in 20 minutes. For larger scale factors it is recommended to use a true cluster; SF=1000 can be generated within 2 hours with 10 such machines connected with Gigabit ethernet (see Figure 3(b)).

[3]When generating URIs that identify entities, we ensure that URIs for the same kind of entity (e.g. person) have an order that follows the time dimension. This is done by encoding the timestamp (e.g. when the user joined the network) in the URI string in an order-preserving way. This is important for URI compression in RDF systems where often a correlation between such identifying URIs and time is present, yet it is not trivial to realize since we generate data in correlation dimension order, not logical time order.



(a) (b)

Figure 3: (a) Friendship degree distribution for scale factor 10. (b) DATAGEN scale-up.

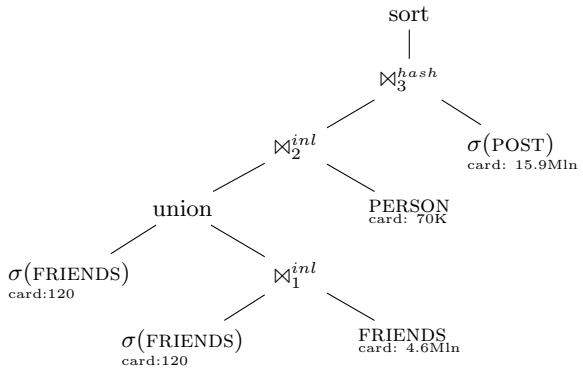


Figure 4: Intended Execution Plan for Query 9

3. DESIGN BY CHOKE POINTS

LDBC benchmark development is driven by the notion of a choke point. A choke point is an aspect of query execution or optimization which is known to be problematical for the present generation of various DBMS (relational, graph and RDF). Our inspiration here is the classical TPC-H benchmark. Although TPC-H design was not based on explicitly formulated choke points, the technical challenges imposed by the benchmark's queries have guided research and development in the relational DBMS domain in the past two decades [3]. A detailed analysis of all choke points used to design the SNB Interactive workload is outside the scope of this paper, the reader can find it in [11]. In general, the choke points cover the "usual" challenges of query processing (e.g., subquery unnesting, complex aggregate performance, detecting dependent group-by keys etc.), as well as some hard problems that are usually not part of synthetic benchmarks. Here we list a few examples of these:

Estimating cardinality in graph traversals with data skew and correlations. As graph traversals are in fact repeated joins this comes back at a crucial open problem of query optimization in a slightly more severe form. SNB queries stress cardinality estimation in transitive queries, such as traversals of hierarchies (e.g., made by replies to posts) and dense graphs (paths in the friendship graph).

Choosing the right join order and type. This problem is directly related to the previous one, cardinality estimation. Moreover, there is an additional challenge for RDF systems where the plan search space grows much faster compared to equivalent SQL queries: SPARQL operates over triple patterns, so table scans on multiple attributes in the relational domain become multiple joins in RDF.

Handling scattered index access patterns. Graph traversals

(such as neighborhood lookup) have random access without predictable locality, and efficiency of index lookup is very different depending on the locality of keys. Also, detecting absence of locality should turn off any locality dependent optimizations in query processing.

Parallelism and result reuse. All SNB Interactive queries offer opportunities for intra- and inter-query parallelism. Additionally, since most of the queries retrieve one- or two-hop neighborhoods of persons in the social graph, and the *Person* domain is relatively small, it might make sense to reuse results of such retrievals across multiple queries. This is an example of recycling: a system would not only cache final query results, but also intermediate query results of a "high value", where the value is defined as a combination of partial query result size, partial query evaluation cost, and observed frequency of the partial query in the workload.

Example. In order to illustrate our choke point-based design of SNB queries, we will describe technical challenges behind one of the queries in the workload, Query 9. Its definition in English is as follows:

Query 9: *Given a start Person, find the 20 most recent Posts/Comments created by that Person's friends or friends of friends. Only consider the Posts/Comments created before a given date.*

This query looks for paths of length two or three, starting from a given Person, moving to the friends and friends of friends, and ending at their created Posts/Comments. This *intended query plan*, which the query optimizer has to detect, is shown in Figure 4. Note that for simplicity we provide the plan and discussion assuming a relational system. While the specific query plan for systems supporting other data models will be slightly different (e.g., in SPARQL it would contain joins for multiple attributes lookup), the fundamental challenges are shared across all systems.

Although the join ordering in this case is fairly straightforward, an important task for the query optimizer here is to detect the types of joins, since they are highly sensitive to cardinalities of their inputs. The lower most join $\bowtie_1$ takes only 120 tuples (friends of a given person) and joins them with the entire FRIENDS table to find the second degree friends. This is best done by looking up these 120 tuples in the index on the primary key of FRIENDS, i.e. by performing an *index nested loop join*. The same holds for the next $\bowtie_2$, since it looks up around a thousand tuples in an index on primary key of PERSON. However, the inputs of the last $\bowtie_3$ are too large, and the corresponding index is not available in POST, so Hash join is the optimal algorithm here. Note that picking a wrong join type hurts the performance here: in the HyPer database system, replacing index-nested loop with hash in $\bowtie_1$ results in 50% penalty, and similar effects are observed in the Virtuoso RDBMS.

Determining the join type in Query 9 is of course a consequence of accurate *cardinality estimation* in a graph, i.e. in a dataset with power-law distribution. In this query, the optimizer needs to estimate the size of second-degree friendship circle in a dense social graph.

Finally, this query opens another opportunity for databases where each stored entity has a unique synthetic identifier, e.g. in RDF or various graph models. There, the system may choose to assign identifiers to POSTS/COMMENTS entities such that their IDs are increasing in time (creation time of the post). Then, the final selection of Posts/Comments created before a certain date will have high locality. Moreover, it will eliminate the need for sorting at the end.

4. SNB-INTERACTIVE WORKLOAD

The SNB-Interactive workload consists of 3 query classes:

Transactional update queries. Insert operations in SNB are generated by the data generator. Since the structure of the SNB dataset is complex, the driver cannot generate new data on-the-fly, rather it is pre-generated. DATAGEN can divide its output in two parts, splitting all data at one particular timestamp: all data before this point is output in the requested bulk-load format (e.g., CSV), the data with a timestamp after the split is formatted as input files for the query driver. These become inserts that are "played out" as the transactional update stream. There are the following types of update queries in the generated data: add a user account, add friendship, add a forum to the social network, create forum membership for a user, add a post/comment, add a like to a post/comment.

Complex read-only queries. The 14 read-only queries shown in the Appendix retrieve information about the social environment of a given user (one- or two-hop friendship area), such as new groups that the friends have joined, new hashtags that the environment has used in recent posts, etc. Although they answer plausible questions that a user of a real social network may need, their complexity is typically beyond the functionality of modern social network providers due to their online nature (e.g., no pre-computation). These queries present the core of query optimization choke points in the benchmark. We have already discussed some of the challenges included in Query 9 in Section 3; the analysis of the rest of the queries is given in [11]. The base definition of the queries is in English, from the LDBC website[4] one can find query definitions in SPARQL, Cypher and SQL, as well as API reference implementations for Neo4j and Sparksee.

Simple read-only queries. The bulk of the user queries are simpler and perform lookups: *(i)* Profile view: for a given user returns basic information from her profile (name, city, age), and the list of at most 20 friends and their posts. *(ii)* Post view: for a given post return basic stats (when was it submitted?) and some information about the sender.

We connect simple with complex read-only queries using a random walk: results of the latter queries (typically a small set of users or posts) become input for simple read-only queries, where *Profile* lookup provides an input for *Post* lookup, and vice versa. This chain of operations is governed by two parameters: the probability to pick an element from the previous iteration P, and the step Δ with which this probability is decreased at every iteration. Clearly, since the probability to continue lookups decreases at each step, the chain will be finite.

Query Mix. Constructing the overall query mix involves defining the number of occurrences of each query type. While doing so, we have two goals in mind. First, the overall mix has to be somewhat realistic. In a social network, this means the workload is read-dominated: for instance, Facebook recently revealed that for each 500 reads there is 1 write in their social network [15]. Second, the workload has to be challenging for the query engine, and consequently the throughput on complex read-only queries should determine a significant part of the benchmark score.

[4] ldbcouncil.org/developer/snb

Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14
132	240	550	161	534	1615	144	13	1425	217	133	238	57	144

Table 4: Frequency of complex read-only queries (number of updates for each query type)

When calibrating SNB-Interactive query mix we aimed at 10% of total runtime to be taken by update queries (taken from the data generator), 50% of time take complex read-only queries, and 40% for the simple read-only queries. Within the corresponding shares of time, we make sure each query type takes approximately equal amount of CPU time (i.e., queries that touch more data run less frequently) to avoid the workload being dominated by a single query. Since updates are given by the data generator, the definition of the query mix is done by setting relative frequencies of read queries (e.g., Query 1 should be performed once in every 132 update operation). The calibration (setting the relative frequencies to fit the target runtime distribution) was performed with Virtuoso RDBMS using explicit plans. In addition, the probability P and the step Δ that control the amount of short reads were also determined experimentally for each supported scale factor. We provide the frequencies of complex read-only queries in Table 4 (see also [5]).

Scaling the workload. If D is the average out-degree of a node in the social graph, and n is the number of entities in the dataset (users/posts/forums), then the 14 read-only queries have complexities $O(D \log n)$, $O(D^2 \log n)$ or $O(D^3 \log n)$, depending on whether they touch one-, two- or three-hop friendship circle. The logarithmic component there is a result of a corresponding index lookup. In contrast, simple read-only and the update queries – all requiring only point lookups in the indexes – are of $O(\log n)$ complexity. Hence, as the dataset increases, our read queries become more "heavy" relatively to updates and short reads. In order to keep the target CPU distribution (10% writes, 40% lookups, 50% reads) as the workload scales, we adjust the frequency of read queries correspondingly (reduce them by the logarithmic factor as the scale factor grows).

Rules and Metrics. Since the scope of our benchmark in terms of systems is very broad, we do not pose any restrictions on the way the queries are formulated. In fact, the preliminary results presented below were achieved by a native graph store (no declarative query language, queries formulated as programs using API) and a relational database system (queries in SQL with vendor-specific extensions for graph algorithms). Moreover, usage of materialized views (or their equivalents) is not forbidden, as long as the system can cope with updates. We require that all transactions have ACID guarantees, with serializability as a consistency requirement. Note that given the nature of the update workload, systems providing snapshot isolation behave identically to serializable.

Our workload contains operations with timestamps in the *simulation time*: updates coming from the data generator, and the read queries that were added according to predefined relative frequencies, as shown in Table 4. A system may be able to execute the workload faster in real time; for example, one hour of simulation time worth of operations might be played against the database system in half an hour of real time. The system under test (SUT) in this situation accepts operations at a certain preset rate, a chosen multiple of the rate in the timeline of the dataset. This *acceleration-factor* (simulation time/real time) that the system can sustain correlates with with throughput of the system.

In order to produce results, a vendor picks a scale of the dataset and the acceleration factor. The run is successful if the system can maintain a steady state throughput compatible with the acceleration factor (simulation time/real time) that was set at the start of the run. Additionally, it is required that latencies of the complex read-only queries are stable as measured by a maximum latency on the 99th percentile. These latencies are reported as a result of the run. Hence the metrics produced by the benchmark are this acceleration-factor and the acceleration-factor/$, i.e. divided by total system cost over 3 years. The cost of the system include hardware and software costs, but not the people costs (that would make price computation extremely vague and location-dependent for the "in-house" solutions). Currently, LDBC allows benchmark runs to be performed in the cloud, but we we extrapolate the operating expenses from the measurement interval to the three year interval, given that inside the measurement interval the workload is uniformly busy. Since people costs are not included into the benchmark score, the cloud runs may be somewhat disadvantaged. We therefore anticipate that there will be a separate category for the cloud-based runs, incomparable with the standalone ("in-house") solutions.

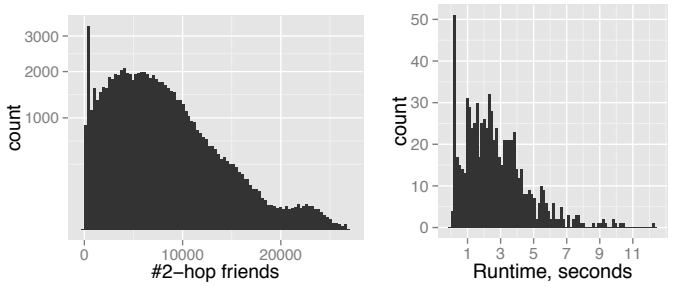
4.1 Innovative Parameter Generation

Motivation and Examples. The benchmark specification provides templates of queries with parameters (e.g., *PersonID*, *Timestamp*, etc.) that are substituted with bindings from the corresponding domain (e.g., all persons or timestamps). Having multiple parameter bindings instead of just one prevents the system from trivially caching the single query results, and it also ensures that a significant portion of the dataset will be touched by the benchmark run. A conventional way to pick parameters is to generate a uniform random sample of the parameter domain, and use values of that sample as parameter bindings. This approach has been employed, among others, by TPC-H and BSBM.

However, selecting uniform random samples from the domain only works well if the underlying values are uniformly distributed and uncorrelated. This is clearly not the case for the LDBC SNB dataset: for example, the distribution of the size of 2-hop environment (i.e., friends and friends of friends) in the SNB graph, depicted in Figure 5a. Since the number of friends has a power-law distribution, the number of friends of friends follows a multimodal distribution with several peaks. Consider now LDBC Query 5 that finds new groups that friends and friends of friends of a given user have joined recently. The uniform sample of *PersonID* for LDBC Query 5 leads to non-uniform distribution of that query runtime (shown in Figure 5b), since the size of the 2-hop environment varies a lot across the users. What is worse, the runtime distribution has a very high variance: there is more than 100 times difference between the smallest and the largest runtime for this sample.

High runtime variance is especially unfortunate, since it leads to non-repeatable benchmark results: by obtaining several uniform samples from the parameter domain (i.e., by running the benchmark several times) we would get very different average runtimes and therefore different scores for the same DBMS, data scale factor and hardware setup.

A similar effect was observed in the TPC-DS benchmark,



(a) Distribution of size of 2-hop friend environment (SNB SF10) (b) Query 5 runtime distr.

Figure 5: Correlations cause high runtime variance (Q5)

where some values have the step-function distribution. TPC-DS circumvents undesired effects by always selecting parameters with the same value of step function (i.e., from the same "step"). However, this trick becomes impossible when the distribution is more complex such as a power-law distribution, and when there are correlations across joins (structural correlations).

In general, in order for the aggregate runtime to be a useful measurement of the system's performance, the selection of parameters for a query template should guarantee the following properties of the resulting queries:

P1: the query runtime has a bounded variance: the average runtime should correspond to the behavior of the majority of the queries

P2: the runtime distribution is stable: different samples of (e.g., 10) parameter bindings used in different query streams should result in an identical runtime distribution across streams

P3: the optimal logical plan (optimal operator order) of the queries is the same: this ensures that a specific query template tests the system's behavior under the well-chosen technical difficulty (e.g., handling voluminous joins, or proper cardinality estimation for subqueries)

It might seem that the ambition in SNB to include queries that are affected by structure/value correlations goes counter to P3, because due to such correlation a particular selection predicate value might for instance influence a join hit ratio in the plan, hence the optimal query plan would vary for different parameter bindings, and picking the right plan would be part of the challenge of the benchmark. Therefore, whenever a query contains correlated parameters we identify *query variants* that correspond to different query plans. For each query variant, though, we would like to obtain parameter bindings with very similar characteristics, i.e. we still need parameter curation.

There are two considerations taken into account when designing the procedure to pick parameters satisfying properties *P1-P3*. First, there is a strong correlation between the runtime of a query and the amount of intermediate results produced during the query execution, denoted C_{out} [9]. Second, as we design the benchmark, we have a specific (*intended*) query plan for each query. For example, LDBC Query 5 mentioned above has an intended query plan as given in Figure 6a. It should be executed by first looking up the person with a given *PersonId*, then finding her friends and friends of friends, and then going through the forums to filter out those that all these friends joined after a certain date. It is therefore sufficient to select parameters with similar runtime for the given query plan.

Now, the problem of selecting (*curating*) parameters from the corresponding domain P with properties *P1-P3* can be formalized as follows:

Parameter Curation: for the Intended Query Plan QI and the parameter domain P, select a subset $S \subset P$ of size k such that $\sum_{\forall T_{qi} \in QI}$ $Variance_{\forall p \in S}$ $C_{out}(T_{qi}(p))$ is minimized. This problem definition requires that the total variance of the intermediate results, taken for every subplan T_{qi} of the plan QI, is minimized across the parameter domain P (in case of multiple parameters P is a cross-product of the respective domains). Since the cost function correlates with runtime, queries with identical optimal plans w.r.t. C_{out} and similar values of the cost function are likely to have close-to-normal distribution of runtimes with small variance.

From the computational complexity point of view, the Parameter Curation problem is not trivial. Intuitively, an exact algorithm would need to tackle a problem which is inverse to the NP-hard *join ordering problem*: for the given optimal plan find the parameters (i.e., queries) which yield a given cost function value. Clearly, we can only seek a heuristic method to solve this at scale.

Note that, as opposed to estimates of C_{out} (that could be obtained from an EXPLAIN feature), we use the de facto amounts of intermediate result cardinalities (which are otherwise only known after the query is executed).

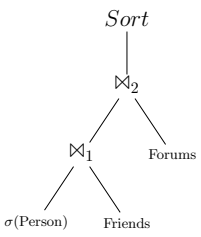
Parameter Curation at scale. Our heuristic for scalable *Parameter Curation* works in two steps:

Step 1: Preprocessing The goal of this stage is to compute all the intermediate results in the query plan for each value of the parameter. We store this information as a Parameter-Count (PC) table, where rows correspond to parameter values, and columns to a specific join result sizes.

As an example, consider LDBC Query 2, which extracts 20 posts of the given user's friends ordered by their timestamps, following the intended plan depicted in Figure 6a. The Parameter-Count table for this query is given in Figure 6b, where columns named $| \bowtie_1 |$ and $| \bowtie_2 |$ correspond to the amount of intermediate results generated by the first and second join, respectively. In other words, when executed with $\%PersonID = 1542$, Query 2 generates $60 + 99 = 159$ intermediate result tuples.

There are two ways to obtain the Parameter-Count table for the entire domain of *PersonID* in our example:
(i) we can form multiple Group-By queries around each subquery in the intended query plan. In our example these are the queries $\Gamma_{PersonID}(Person \bowtie Friend)$ and $\Gamma_{PersonID}((Person \bowtie Friend) \bowtie Forum)$. The result of these queries are first and second column in Parameter-Count table, respectively. Or, alternatively
(ii) since we are generating the data anyway, we can keep the corresponding counts (number of friends per user and number of posts per user) as a by-product of data generation. SNB-Interactive uses this strategy: DATAGEN in a final stage curates parameters based on frequency statistics.

The Parameter-Count table needs to be materialized for every query template. While it is feasible for discrete parameters with reasonably small domains (like *PersonID* in



(a) Intended Plan

PersonID	$\lvert\rhd\lhd_1\rvert$	$\lvert\rhd\lhd_2\rvert$
...	...	...
1542	60	99
1673	60	102
7511	60	103
958	60	120
1367	61	101
...	...	...

(b) Parameter-Count table

Figure 6: Parameter Curation for Query 2

SNB dataset), it becomes too expensive for continuous parameters. In that case, we introduce *buckets* of parameters (for example, group *Timestamp* parameter into buckets of one month length), see [6] for more details.

Step 2: Greedy parameter selection Once the intermediate results for the query template are computed, our Parameter Curation problem boils down to finding *similar* rows (i.e., with the smallest variance across all columns) in the Parameter-Count table. Here we rely on a greedy heuristics that forms *windows* of rows with the smallest variance. In our example Figure 6b we first identify the windows of rows in the column $\mid \bowtie_1 \mid$ with the minimum variance (depicted with dark gray color). Then, in this window we find the sub-window with the smallest variance in the second column $\mid \bowtie_2 \mid$. This procedure continues on further columns (if present). In our example the initial window on the first column consists of rows with count 60, and among these rows we pick rows with values 99, 102, and 103 in the second column (dark gray color in Figure 6b). These rows correspond to bindings 1542, 1673 and 7511 of *PersonID*. At the end, every initial window on the first column is refined to contain rows with the smallest variance across all columns. We use the corresponding *PersonID* from these rows (across the entire Parameter-Count table) to collect the required k parameter bindings.

Parameter Curation for multiple parameters. The procedure described above can be easily generalized to the case of multiple parameters [6]. In particular, we have used it for picking parameters in the following two situations that occur in LDBC SNB queries: *1) A query with two (potentially correlated) parameters*, one from discrete and another from continuous domain, such as *Person* and *Timestamp* (of her posts, orders, etc). *2) Multiple (potentially correlated) parameters*, such as *Person*, her *Name* and her *Country*.

4.2 Workload Driver

Traditionally, a transactional workload is split into partitioned (*streams*) that are issued concurrently against the System Under Test in order to get maximal throughput. In case of SNB-Interactive, splitting update operations into parallel streams is not trivial, since updates may depend on each other: a user can not add a friendship before the corresponding friend profile is created (these two operations may be in two different streams), a comment can be added only to existing post, etc. Some parts of the update workload can be easily partitioned: for example, updates touching posts/-comments from one forum are assigned to the same update stream. On the other hand, any update that takes *PersonId* potentially touches the `FRIEND` graph, which is non-partitionable. The negative consequence would be running only a single stream, or multiple streams where the clients must synchronize their activities. Both alternatives could severely limit the throughput achieved by the query driver.

Tracking Dependencies. To have greater control over the generated load profile (e.g., to generate trending topics, see Figure 2b) every operation in a workload has a timestamp, referred to as Due Time (T^{DUE}), which represents the simulation time at which that operation is scheduled to be executed.

In addition, each operation belongs to none, one, or both of the following sets: *Dependencies* and *Dependents*. *Dependencies* contains operations that introduce dependencies in the workload (e.g., create profile); for every operation in this set there exists at least one other operation (from *Dependents*) that can not be executed until this operation has completed execution. *Dependents* contains operations that are dependent on at least one other operation (from *Dependencies*) in the workload, for instance, adding a friend. The driver uses this information when tracking inter-operation dependencies, to ensure they are not violated during execution. It tracks the latest point in time behind which every operation has completed; every operation (i.e., dependency) with T^{DUE} lower or equal to this time is guaranteed to have completed execution. This is achieved by maintaining a monotonically increasing timestamp variable called Global Completion Time (T^{GC}), which every parallel stream has access to. Every time the driver begins execution of a *Dependencies* operation the timestamp of that operation is added to Initiated Times (IT): set of timestamps of operations that have started executing but not yet finished. Upon completion, timestamps are removed from IT and added to Completed Times (CT): set of timestamps of completed operations. Timestamps must be added to IT in monotonically increasing order but can be removed in any order.

More specifically, dependency tracking is performed as follows. Each stream has its own instances of IT and CT which, along with the dependency-tracking logic, are encapsulated in Local Dependency Service (LDS); its data structures and logic are given in Figure 7. As well as maintaining IT and CT, LDS exposes two timestamps: Local Initiation Time (T^{LI}) and Local Completion Time (T^{LC}). T^{LI} is the lowest timestamp in IT, or the last known lowest timestamp if IT is empty. T^{LC} is a local analog to T^{GC}, the point in time behind which every operation from that particular stream has completed; there is no lower or equal timestamp in IT and at least one equal or higher timestamp in CT. T^{LI} and T^{LC} are guaranteed to monotonically increase.

Inter-stream dependency tracking is performed by Global Dependency Service (GDS) similarly to how LDS tracks intra-stream dependencies, but instead of internally tracking IT and CT it tracks LDS instances. Like LDS, GDS exposes two timestamps: Global Initiation Time (T^{GI}) and T^{GC}. T^{GI} is the lowest T^{LI} from across all LDS instances. T^{GC} is the point in time behind which every operation, from all streams, has completed; there is no LDS with T^{LI} lower or equal to this value and at least one LDS has T^{LC} equal or higher than this value. T^{GI} and T^{GC} are guaranteed to monotonically increase.

The rationale for exposing T^{LI} is that, as values added to IT are monotonically increasing, T^{LI} communicates that no lower value will be submitted in the future, enabling GDS to advance T^{GC} as soon as possible. Strictly, T^{GI} is not required in a single process context. The rationale for exposing

```
class LocalDependencyService {
  Time[] IT
  Time[] CT
  Time T^LI <- max(T^LI, min([i for i in IT]))
  Time T^LC <- max([c for c in CT: c<T^LI])
}

class GlobalDependencyService {
  LocalDependencyService[] LDS
  Time T^GI <- min([l.T^LI for l in LDS])
  Time T^GC <- max([l.T^LC for l in LDS: l.T^LC<T^GI])
}
```

Figure 7: Dependency tracking classes

T^{GI} is to make *GDS* composable. That is, a *GDS* instance could track other *GDS* instances in the same manner as it tracks *LDS* instances, enabling dependency tracking in a hierarchical/distributed setting.

Stream Execution Modes. Every operation, regardless of dependencies, is executed in a similar manner, illustrated in Figure 8. The default *Execution Mode* (method of scheduling operations) is *Parallel*: multiple stream operations are executed in parallel, using a thread pool, and the dependencies are satisfied using T^{GC} communication. However, for some types of operations it is possible to use a simple *Sequential* execution mode, where very limited communication between driver threads is necessary, since *most of the dependencies stay within one stream*. In this section we describe the motivation and applicability of this *Sequential mode* to update execution in SNB-Interactive.

Some dependencies are difficult to capture efficiently with T^{GC} alone. For example, consider a subset of the SNB-Interactive workload: the creation of users, posts, and likes. Likes depend on the existence of posts, posts and likes depend on the existence of the users that created them. Users are created at a much lower frequency than posts and likes, and do not immediately create content. Conversely, posts are replied to and/or liked soon after their creation.

Using T^{GC} to maintain dependencies between posts and likes would result in many frequent updates to T^{GC}, and excessive synchronization between streams as they wait for T^{GC} to advance. However, observe that posts and likes form a tree, rooted at the forum, therefore it is possible to partition update streams by forum, eliminating inter-forum dependencies. The insight here is that posts and likes only depend on other posts from the same forum, as long as intra-forum dependencies are maintained, updates to a given forum can progress irrespective of the state of other forums.

Moreover, when dependent operations occur at high frequency (duration between T^{DUE} of dependent operations is short) the benefit of parallel execution might be negated by the cost of dependency tracking in the query driver. The alternative offered by *Sequential* execution is that instead of classifying stream operations as *Dependent*/*Dependency*, the same dependencies can be captured by executing that stream sequentially, thereby guaranteeing causal order is maintained. This, however, only applies when it is possible to partition streams into many smaller streams, to achieve sufficient parallelism.

In the SNB-Interactive case *Sequential* execution is used for capturing intra-forum dependencies - using T^{GC} would introduce false dependencies. This dramatically reduces overhead related to dependency tracking, and achieves sufficient parallelism due to the large number of forums.

```
Operation operation <- stream.next()
if (dependencies.contains(operation)){
  LDS.IT.add(operation.DUE)
}
if (dependents.contains(operation)){
  while(operation.DEP < GDS.GCT){
    // wait
  }
}
while(operation.DUE < now()){
  // wait
}
operation.execute()
if (dependencies.contains(operation)){
  LDS.IT.remove(operation.DUE)
  LDS.CT.add(operation.DUE)
}
```

Figure 8: Dependent execution

For dependencies between users and their generated content T^{GC} tracking is used, as it is impossible to partition the social graph in such a way that dependencies are eliminated.

Windowed Execution. The mechanisms we introduced so far guarantee that dependency constraints are not violated, but in doing so they unavoidably introduce overhead of synchronization between driver threads.

In so-called *Windowed Execution* mode, operations are executed in groups (*Windows*), where operations are grouped according to their T^{DUE}. Every *Window* has a Start Time, Duration, and End Time. Logically, all operations in a *Window* are executed at the same time, some time within the *Window*. No guaranty is made regarding exactly when, or in what order, an operation will execute within its *Window*. Operations belonging to *Dependencies* are never executed in this manner – T^{DUE} of *Dependencies* operations are never modified as it would affect how dependencies are tracked.

To allow *Windowed Execution* mode we must ensure a minimum duration exists between the T^{DEP} and T^{DUE} of any operation in *Dependents*, this is called "Safe Time" (T^{SAFE}). In the case of the SNB dataset, what we need to know is the minimum time between a person becoming a member of the network and making a first post, and a minimum time between becoming a friend and writing a first comment or like in the friend's forum. DATAGEN ensures that this T^{SAFE} is considerably long in all generated data. The end effect of *Windowed Execution* is that the T^{GC} between the parallel threads (or processes) in the driver need to be synchronized much less often (once every T^{SAFE} of simulated time). This helps reduce communication overhead, and this mode also gives threads the ability to schedule queries inside a window out-of-order and hence be less bursty.

In the currently available version of the driver, which is multi-threaded but single-node, *Windowed Execution* mode is not yet available. We plan to make this available in the multi-node version of the driver, where synchronization cost would be high (as it involves network communication).

Scalable Dependent Execution. To illustrate driver scalability, experiments were performed using a dummy database connector that, rather than executing transactions against a database, simply sleeps for a configured duration. From the driver perspective this simulates a benchmark run where the SUT takes, on average, that duration to execute a transaction. The experiment was run with two configured sleep durations: 1ms and 100us.

The chosen workload consists only of the SNB-Interactive updates. Specifically, all updates from SF10 update stream

partitions:	1	2	4	8	12
1ms	997	1990	3969	7836	11298
100us	9745	19245	38285	78913	110837

Table 5: Op/second vs #partitions

complex read-only	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14
Sparksee,SF10	20	44	441	31	100	41	11	38	3376	194	66	177	794	2009
Virtuoso,SF300	941	1493	4232	1163	2688	16090	1000	32	18464	1257	762	1519	559	742

Table 6: Mean runtime of complex read-only queries (ms)

simple read-only	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Sparksee,SF10	7	9	9	8	9	9	8
Virtuoso,SF300	6	147	37	7	2	1	8

Table 7: Mean runtime of simple read-only queries (ms)

- approximately 32 Million operations. Note that, as they contain no inter-dependencies, executing the read queries in parallel is trivial and uninteresting from the perspective of driver scalability. To control parallelism, the number of partitions was set from 1 to 12. All experiments were done on a system with 12 Intel Xeon E5-2640 CPUs, 128GB RAM, and SSD storage running Linux's 3.2.0-57 kernel.

As presented in Table 5, the driver shows near-linear scalability while maintaining the complex inter-partition dependencies, i.e., ensuring dependent operations do not start until their dependencies complete. Every entry in Table 5 corresponds to the execution of the exact same stream (in SF10 the stream comprises of 32,648,010 forum operations and 6,889 user operations, which are spread uniformly across the timeline), the only differences being the number of partitions the stream is divided into. As partition count (parallelism) increases so does the execution rate, reducing the time between subsequent operations, in turn straining more the tracking of dependencies. Remembering that every forum operation depends on one user operation, this increased throughput translates to a greater probability for any of the forum operations to block as they wait for their user dependency to complete. Further, because forums are executed using synchronous execution mode, the blocking of one operation would result in the blocking of an entire partition.

5. EVALUATION

We report some results of running SNB-Interactive with two systems: Sparksee (native graph database) and Virtuoso (hybrid relational/RDF store). All the runs were performed on the same machine, a single dual Xeon E5 2630 with 192GB of RAM and 6 magnetic disks. It should be noted that both vendors are still in the process of tuning their systems and these results are preliminary.

Scale Factor 10: Sparksee First we run the Interactive workload on SF10 dataset with Sparksee graph database. The queries were implemented using Sparksee's Java API. Sparksee creates indexes on IDs of nodes, and additionally materializes neighborhood of each node. Our acceleration ratio for this run is 0.1, measured throughput is around 26 operations per second. For the 10GB dataset, the Sparksee image takes 27GB; the execution is therefore fully in memory. We provide the mean latencies of complex read queries in Table 6. Results of short read queries and transactional updates are given in Tables 7 and 9, respectively.

Scale Factor 300: Virtuoso We run the SNB SF300 on Openlink Virtuoso 7.5. The benchmark implementation is in SQL using Virtuoso transitive SQL extensions for graph traversals. The tables are column-wise compressed and indices are created on foreign key columns where needed, otherwise all is in primary key order. The 300GB dataset is 88GB after gzip compression.

In Table 8 we give the sizes in MB of allocated database pages for three largest tables and their largest indices, loaded into Virtuoso Column store. 138GB is the total allocated space including all column and row-wise structures. We note that both identifiers and datetimes compress significantly from their CSV form.

The benchmark was run with acceleration of 10 units of simulation time per 4 of real time (0.4), reaching a throughput of 500 queries per second. Table 6 gives the number of executions and mean client-side duration of each query. The achieved throughput would be plausible as a peak load of an online system, as these are usually sized to run at a fraction of theoretical peak throughput. Thus, a 300GB dataset with 1.1 million people could be served from a small/medium commodity server (12 core, 192GB RAM). Conversely, a 100 million people network would take over 200 servers in a redundant cluster configuration, which is again plausible. Tables 7 and 9 have results of short read queries and transactional updates of the Virtuoso SF-300 run. More experimental results for Virtuoso can be found in [12].

6. RELATED WORK

Benchmarking has become popular in the RDF/Semantic Web community, partially because there is a standard query language, SPARQL. Consequently many benchmarks exist in the area, including LUBM, BSBM, and SP^2Bench. Although these benchmarks often cover many features of SPARQL, even BSBM, a rather advanced benchmark, does not reach the classical TPC-H in terms of query optimization challenges. SNB-Interactive workload, on the other hand, follows a *query language-independent*, *choke point-based* approach to provide challenges for modern DBMS.

Related to social network benchmarking, Facebook recently presented Linkbench [1], a benchmark targetting the OLTP workload on the Facebook graph. It is, however, rather limited in scope (only transactions) and uses a synthetic graph generator that, besides degree distribution, reproduces very little of the structure or value correlations found in real networks. In contrast, SNB's DATAGEN provides the ground for multiple realistic workloads ranging from OLTP to graph algorithms.

The BG benchmark [2] proposes to evaluate simple social networking actions under different Service Level Agreements. We note that LDBC queries are more complex than BG, and require more stict consistency requirements (ACID). Further, it would be impossible to validate benchmark runs for such relaxed consistency models.

In the super-computing domain, we find Graph-500, which consists of Breadth First Search queries, and is used to test the hardware capabilities of large scale systems for work-

Table	Size (MB)	Largest Index (MB)
post	76815	ps_content (41697)
likes	23645	l_creationdate (11308)
forum_person	9343	fp_creationdate (5957)

Table 8: Size of 3 largest tables - Virtuoso,SF300

updates	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Sparksee,SF10	492	309	307	239	317	190	324	273
Virtuoso,SF300	35	198	85	55	16	118	141	15

Table 9: Mean runtime of transactional updates (ms)

loads exhibiting more random memory access patterns than those found in traditional scientific applications.

7. CONCLUSION

The LDBC SNB introduces a new and quite complex synthetic social network dataset on which three workloads are intended to be run: SNB-Interactive, SNB-BI and SNB Algorithms. This paper focuses on the former, which tests on-line queries. The benchmark has been implemented on graph database systems, RDF database systems and RDBMSs. The SNB data generator is innovative due to its power-law driven data generation with realistic correlations between properties and graph structure, as well as its scalable Hadoop implementation – allowing to generate terabytes of data quickly on a small cluster. The SNB query driver needed to confront the issue of non-partitionability of the transaction workload, since social graphs are one huge connected component. The SNB-Interactive query mix is a balance between testing so-called "choke points" with the complex read-only queries, and executing simple updates and read-only queries. A final contribution is the introduction of *parameter curation* which data mines the dataset for query parameters with highly similar behavior, to make the benchmark score more insightful and stable across runs.

SNB-Interactive contains a rich set of technical challenges, of which we are convinced that the current generation of systems only a few target effectively. This makes it an interesting benchmark for IT practitioners, industry engineers and academics alike.

8. REFERENCES

[1] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. LinkBench: A Database Benchmark Based on the Facebook Social Graph. SIGMOD '13, 2013.

[2] S. Barahmand and S. Ghandeharizadeh. Bg: A benchmark to evaluate interactive social networking actions. In *CIDR*, 2013.

[3] P. A. Boncz, T. Neumann, and O. Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *TPCTC*, pages 61–76, 2013.

[4] A. Clauset, C. R. Shalizi, and M. E. Newman. Power-law distributions in empirical data. *SIAM review*, 2009.

[5] A. Gubichev. Benchmarking transactions. `http://ldbc.eu/sites/default/files/LDBC_D2.2.3_final.pdf`.

[6] A. Gubichev and P. Boncz. Parameter curation for benchmark queries. TPCTC'14.

[7] J. Leskovec et al. Meme-tracking and the dynamics of the news cycle. In *SIGKDD*, 2009.

[8] M. McPherson et al. Birds of a feather: Homophily in social networks. *Annual review of sociology*, 2001.

[9] G. Moerkotte. Building Query Compilers. `http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf`.

[10] M.-D. Pham, P. Boncz, and O. Erling. S3G2: a Scalable Structure-correlated Social Graph Generator. In *TPCTC*, 2012.

[11] A. Prat and A. Averbuch. Benchmark design for navigational pattern matching benchmarking. `http://ldbcouncil.org/sites/default/files/LDBC_D3.3.34.pdf`.

[12] A. Prat and A. Averbuch. Benchmark design for navigational pattern matching benchmarking - Benchmark Executions. `http://ldbcouncil.org/sites/default/files/LDBC_D3.3.34_appendix.pdf`.

[13] A. Prat-Pérez and D. Domínguez-Sal. How community-like is the structure of synthetically generated graphs? In *GRADES*, 2014.

[14] J. Ugander et al. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503*, 2011.

[15] J. Wiener and N. Bronson. Facebook's top open data problems. `https://research.facebook.com/blog/1522692927972019/facebook-s-top-open-data-problems/`, 2014.

APPENDIX

Definitions of 14 read queries of the Interactive workload. See [11] for their SQL/SPARQL/Cypher formulations. For each query we additionally highlight its `parameters`.

Q1. *Extract description of friends with a given name* Given a person's `firstName`, return up to 20 people with the same first name, sorted by increasing distance (max 3) from a given `person`, and for people within the same distance sorted by last name. Results should include the list of workplaces and places of study.

Q2. *Find the newest 20 posts and comments from your friends.* Given a start `Person`, find (most recent) Posts and Comments from all of that Person's friends, that were created before (and including) a given `Date`. Return the top 20 Posts/Comments, and the Person that created each of them. Sort results descending by creation date, and then ascending by Post identifier.

Q3. *Friends within 2 steps that recently traveled to countries X and Y.* Find top 20 friends and friends of friends of a given `Person` who have made a post or a comment in the foreign `CountryX` and `CountryY` within a specified period of `DurationInDays` after a `startDate`. Sorted results descending by total number of posts.

Q4. *New Topics.* Given a start `Person`, find the top 10 most popular Tags (by total number of posts with the tag) that are attached to Posts that were created by that Person's friends within a given `time interval`.

Q5. *New groups.* Given a start `Person`, find the top 20 Forums the friends and friends of friends of that Person joined after a given `Date`. Sort results descending by the number of Posts in each Forum that were created by any of these Persons.

Q6. *Tag co-occurrence.* Given a start `Person` and some `Tag`, find the other Tags that occur together with this Tag on Posts that were created by Person's friends and friends of friends. Return top 10 Tags, sorted descending by the count of Posts that were created by these Persons, which contain both this Tag and the given Tag.

Q7. *Recent likes.* For the specified `Person` get the most recent likes of any of the person's posts, and the latency between the corresponding post and the like. Flag Likes from outside the direct connections. Return top 20 Likes, ordered descending by creation date of the like.

Q8. *Most recent replies.* This query retrieves the 20 most recent reply comments to all the posts and comments of `Person`, ordered descending by creation date.

Q9. *Latest Posts.* Find the most recent 20 posts and comments from all friends, or friends-of-friends of `Person`, but created before a `Date`. Return posts, their creators and creation dates, sort descending by creation date.

Q10. *Friend recommendation.* Find top 10 friends of a friend who posts much about the interests of `Person` and little about not interesting topics for the user. The search is restricted by the candidate's `horoscopeSign`. Returns friends for whom the difference between the total number of their posts about the interests of the specified user and the total number of their posts about topics that are not interests of the user, is as large as possible. Sort the result descending by this difference.

Q11. *Job referral.* Find top 10 friends of the specified `Person`, or a friend of her friend (excluding the specified person), who has long worked in a company in a specified `Country`. Sort ascending by start date, and then ascending by person identifier.

Q12. *Expert Search.* Find friends of a `Person` who have replied the most to posts with a tag in a given `TagCategory`. Return top 20 persons, sorted descending by number of replies.

Q13. *Single shortest path.* Given `PersonX` and `PersonY` , find the shortest path between them in the subgraph induced by the Knows relationships. Return the length of this path.

Q14. *Weighted paths.* Given `PersonX` and `PersonY` , find all weighted paths of the shortest length between them in the subgraph induced by the Knows relationship. The weight of the path takes into consideration amount of Posts/Comments exchanged.