LDBC

Cooperative Project

FP7 – 317548

Social Network Benchmark Specification

Coordinator: [Arnau Prat (UPC)]

With contributions from: [Peter Boncz (VUA), Josep Lluís Larriba (UPC), Renzo Angles (TALCA), Alex Averbuch (NEO), Orri Erling (OGL), Andrey Gubichev (TUM), Mirko Spasić (OGL)], Minh-Duc Pham (VUA), Norbert Martínez (SPARSITY)

Deliverable nature:	Report (R)
Dissemination level: (Confidentiality)	Public (PU)
Contractual delivery date:	M12
Actual delivery date:	M13
Version:	0.1
Total number of pages:	78
Keywords:	benchmark, choke points, dataset generator, graph database, query set, RDF, workload, auditing rules, publication rules, scale factors

Abstract

LDBC's Social Network Benchmark (LDBC-SNB) is an effort intended to test various functionalities of systems used for graph-like data management. For this, the recognizable scenario of using graph-shaped data in a large social network is used.

LDBC-SNB consists of three sub-benchmarks, or workloads, that focus on different functionalities. In this document, a preliminary version of the Interactive Workload, which contains small lookup queries, is presented. The other workloads, still in development, are the Business Intelligence Workloads (with analytical queries), and the Graph Analytics Workload (with graph algorithms).

This document contains a detailed explanation of the data used in the whole LDBC-SNB benchmark, a detailed description for all the Interactive Workload lookup queries, and instructions on how to generate the data and run the benchmark with the provided software.

Executive Summary

TODO - Peter, Larri

Document Information

IST Project Number	FP7 – 317548	Acronym	LDBC
Full Title	LDBC		
Project URL	http://www.ldbc.eu/		
Document URL	`https://svn.sti2.at/ldbc/trunk/sib_spec/sib_spec.tex/`		
EU Project Officer	Carola Carstens		

Deliverable	Number		Title	Social Network Benchmark Specification
Work Package	Number	N/A	Title	Social Network Benchmark Task Force

Date of Delivery	Contractual	M12	Actual	M13
Status	version 0.1		final ☐	
Nature	Report (R) ☒ Prototype (P) ☐ Demonstrator (D) ☐ Other (O) ☐			
Dissemination Level	Public (PU) ☒ Restricted to group (RE) ☐ Restricted to programme (PP) ☐ Consortium (CO) ☐			

Authors (Partner)	Arnau Prat (UPC)			
Responsible Author	Name	Arnau Prat	E-mail	aprat@ac.upc.edu
	Partner	UPC	Phone	+34934054032

Abstract (for dissemination)	LDBC's Social Network Benchmark (LDBC-SNB) is an effort intended to test various functionalities of systems used for graph-like data management. For this, the recognizable scenario of using graph-shaped data in a large social network is used. LDBC-SNB consists of three sub-benchmarks, or workloads, that focus on different functionalities. In this document, a preliminary version of the Interactive Workload, which contains small lookup queries, is presented. The other workloads, still in development, are the Business Intelligence Workloads (with analytical queries), and the Graph Analytics Workload (with graph algorithms). This document contains a detailed explanation of the data used in the whole LDBC-SNB benchmark, a detailed description for all the Interactive Workload lookup queries, and instructions on how to generate the data and run the benchmark with the provided software.
Keywords	benchmark, choke points, dataset generator, graph database, query set, RDF, workload, auditing rules, publication rules, scale factors

Version Log			
Issue Date	Rev. No.	Author	Change
19/05/2014	0.1	Arnau Prat	First draft

Table of Contents

List of Figures

List of Tables

Definitions

DBGEN: Is the data set generator provided by the LDBC-SNB, which is responsible of generating the data needed to run the benchmark.

DBMS: A DataBase Management System.

LDBC-SNB: Linked Data Benchmark Council Social Network Benchmark.

Query Mix: Refers to the ratio between read and update queries of a workload, and the frequency at which they are issued.

SF: See Scale Factor.

Scale Factor: The LDBC-SNB is designed to target systems of different size and scale. The scale factor determines the size of the data used to run the benchmark, in terms of Gigabytes.

SUT: The System Under Test is defined to be the database system where the benchmark is executed.

Test Driver: A program provided by the LDBC-SNB, which is responsible of executing the different workloads and gathering the results.

Test Sponsor: The Test Sponsor is the company officially submitting the Result with the FDR and will be charged the filing fee. Although multiple companies may sponsor a Result together, for the purposes of the LDBC processes the Test Sponsor must be a single company. A Test Sponsor need not be a LDBC member. The Test Sponsor is responsible for maintaining the FDR with any necessary updates or corrections. The Test Sponsor is also the name used to identify the Result.

Workload: A workload refers to a set of queries of a given nature (i.e interactive, analytical, business), how they are issued and at which rate.

1 Introduction

1.1 Motivation for the Benchmark

The new era of data economy, based on large, distributed and complexly structured data sets, has brought on new and complex challenges in the field of data management and analytics. These data sets, usually modeled as large graphs, have attracted both the industry and academia, due to the new opportunities in research and innovation they offer. This situation has also opened the door for new companies to emerge, offering new non-relational and graph-like technologies that are called to play a significant role in upcoming years.

The change in the data paradigm, calls for new benchmarks to test the new emerging technologies, as they set a framework where different systems can compete and compare in a fair way, they let technology providers to identify the bottlenecks and gaps of their systems and, in general, drive the research and development of new information technology solutions. Without them, the uptake of these technologies is at risk by not providing the industry with clear, user-driven targets for performance and functionality.

The LDBC Social Network Benchmark (LDBC-SNB) aims at being comprehensive benchmark setting the rules for the evaluation of graph-like data management technologies. LDBC-SNB is designed to be a plausible look-alike of all the aspects of operating a social network site, as one of the most representative and relevant use case of modern graph-like applications. LDBC-SNB is a work in progress, and initially, it only includes a read only Interactive Workloads, but two more workloads will be introduced in the future: the Business Intelligence and the Analytics. All three workloads will support update queries in future releases. By designed three separate workloads, LDBC-SNB targets a broader range of systems with different nature and characteristics. LDBC-SNB aims at capturing the essential features of these usage scenarios while abstracting away details of specific business deployments.

1.2 Relevance to Industry

LDBC-SNB is intended to provide the following value to different stakeholders:

- For **end users** facing graph processing tasks, LDBC-SNB provides a recognizable scenario against which it is possible to compare merits of different products and technologies. By covering a wide variety of scales and price points, LDBC-SNB can serve as an aid to technology selection.
- For **vendors** of graph database technology, LDBC-SNB provides a checklist of features and performance characteristics that helps in product positioning and can serve to guide new development.
- For **researchers**, both industrial and academic, the LDBC-SNB dataset and workload provide interesting challenges in multiple choke-point areas, such as query optimization, (distributed) graph analysis, transactional throughput, and provides a way to objectively compare the effectiveness and efficiency of new and existing technology in these areas.

The technological scope of the LDBC-SNB comprises all systems that one might conceivably use to perform social network data management tasks:

- **Graph database systems** (e.g. neo4j, InfiniteGraph, Sparksee, Titan)
  - these systems store directed, labeled graphs; and support traversals via APIs
  - they often also support a query language (e.g. Gremlin, Cypher), but this may not be the primary interface
  - queries/programs/tasks programmed against the graph data often involve updating a state specific to the task attached to potentially all nodes/edges
  - these systems often support value-based indexes to quickly locate nodes/edges

  - these systems often support transactional queries, with some degree of consistency
  - typically single-machine architecture (non-cluster)

- **Graph programming frameworks** (e.g. Giraph, Signal/Collect, Graphlab, Green Marl)
  - the core of this system is a language/API that allow to create graph manipulations focused on successive actions on sets of nodes, executing in parallel or lockstep
  - these systems often interface (or take the form of a library) inside a programming language, such that graph manipulation steps and custom logic are intertwined
  - these frameworks typically target global graph computations
    * with long latency
    * involving many graph nodes/edges
    * which often compute approximation answers of problems that are often NP-complete
  - both single-machine and cluster systems exist

- **RDF database systems** (e.g. OWLIM, Virtuoso, BigData, Jena TDB, Stardog, Allegrograph)
  - these systems implement the SPARQL1.1 query language similar in complexity to SQL1992, which allows for structured queries, and simple traversals
  - RDF database system often come with additional support for simple reasoning (sameAs,subClass) , text search and geospatial predicates
  - RDF database systems generally support transactions, but not always with full concurrency and serializability
  - RDF database systems supposed strength is integrating multiple data sources (e.g. DBpedia)
  - both single-machine and cluster systems exist

- **Relational database systems** (e.g. Postgres, MySQL, Oracle, DB2, SQLserver, Virtuoso, MonetDB, Vectorwise, Vertica, but also Hive and Impala)
  - data is relational, and queries are formulated in SQL and/or PL/SQL
  - both single-machine and cluster systems exist
  - relational systems do not normally support recursion, or stateful recursive algorithms, which makes them not at home in the graph analytics workloads

- **noSQL database systems** (e.g. key-value stores such as HBase, REDIS, MongoDB, CouchDB, or even MapReduce systems like Hadoop and Pig).
  - all these systems are cluster-vbased and scalable.
  - the key-value stores could possibly implement the Interactive Workload, though its navigational aspects would pose some problems as potentially many key-value lookups are needed.
  - the MapReduce systems could be suited for the Graph Analytics workload.
  - Pure MapReduce would probably have query latency that is so high that the Business Intelligence workload would not make sense, though we note that some of the key-value stores (e.g. MongoDB) provide a MapReduce query functionality on the data that it stores which could make it suited for the BI workload.

We can further add to this list noSQL database system (HBase, Redis, MongoDB, CouchDB, etc), though such systems would most likely be restricted to handling the Interactive Workload. Those systems with a MapReduce based query functionality could possibly also

1.3 General Benchmark Overview

LDBC-SNB aims at being a complete benchmark, designed with the following goals in mind:

- **Rich coverage**. LDBC-SNB is intended to cover most demands encountered in the management of complexly structured data.
- **Modularity**. LDBC-SNB is broken into parts that can be individually addressed. In this manner LDBC-SNB stimulates innovation without imposing an overly high threshold for participation.
- **Reasonable implementation cost**. For a product offering relevant functionality, the effort for obtaining initial results with SNB should be small, on the order of days.
- **Relevant selection of challenges**. Benchmarks are known to direct product development in certain directions. LDBC-SNB is informed by the state of the art in database research so as to offer optimization challenges for years to come while not having a prohibitively high threshold for entry.
- **Reproducibility and documentation of results**. LDBC-SNB will specify the rules for full disclosure of benchmark execution and for auditing of benchmark runs. The workloads may be run on any equipment but the exact configuration and price of the hardware and software must be disclosed.

LDBC-SNB benchmark is modeled around the operation of a real social network site. A social network site represents a relevant use case for the following reasons:

- It is simple to understand for a large audience, as it is arguably present to our every-day life in different shapes and forms.
- It allows testing a complete range of interesting challenges, by means of different workloads targeting systems of different nature and characteristics.
- A social network can be scaled, allowing the design of a scalable benchmark targeting systems of different sizes and budgets.

In Section 2.2, LDBC-SNB defines the schema of the data used in the benchmark. The schema, represents a realistic social network, including people and their activity in the social network during a period of time. Personal information of each person, such as the name, the birth day, interests or the places where people work or study, is included. Persons' activity is represented in the form of friendhisp relationships and content sharing (i.e messages and pictures). LDBC-SNB provides a scalable synthetic data generator based on the MapReduce parallel paradigm, that produces networks with the described schema with distributions and correlations similar to those expected in a real social network. Furthermore, the data generator is designed to be user friendly. The proposed data schema is shared by all the different proposed workloads, those we currently have, and those that will be proposed in the future.

In Section 2.3, the Interactive Workload is proposed. Currently it only defines read queries, but will be updated in the near future to support updates. Two more workloads are planned: Business Intelligence Workload and Analytical workload. Workloads are designed to mimic the different usage scenarios found in operating a real social network site, and each of them targets one or more types of systems. Each workload defines a set of queries and query mixes, designed to stress the SUTs in different choke-point areas, while being credible and realistic. Interactive workload reproduces the interaction between by the users of the social network, including lookups and transactions that update small portions of the data base. These queries are designed to be interactive and target systems capable of responding such queries with low latency for multiple concurrent users. Business Intelligence workload, will represents those business intelligence analytics a social network company would like perform in the social network, in order to take advantage of the data to discover new business opportunities. This workload explore moderate portions of data from different entities, and performing more resource intensive operations. Finally, the graph analytics workload will aim at exploring the characteristics of the underlying

structure of the network. Shortest paths, community detection or centrality, are representative queries of this workload, and will imply touching a vast amount of the dataset.

LDBC-SNB provides an execution test driver, which is responsible of executing the workloads and gathering the results. The driver is designed with simplicity and portability in mind, to ease the implementation on systems with different nature and characteristics, at a low implementation cost. Furthermore, it will automatically handle the validation of the queries in the near future. The overall philosophy of LDBC-SNB is to provide all the necessary software tools to run the benchmark, and therefore to reduce the benchmark's entry point as much as possible.

A complete benchmark needs to define a set of performance metrics for each workload, to allow a fair comparison of systems of different sizes and budgets. We understand that different workloads require different performance metrics, and that these have to capture not only the pure performance of the SUT, but also its performance per price. LDBC-SNB is aware of that and will provide performance metrics for each of the workloads proposed.

Similarly to performance metrics, LDBC-SNB will define the rules for executing the benchmark. Rules will include the different steps to follow in order to have a valid benchmark execution, as well as those things that are prohibited. The goal is to ensure, as much as possible, that the reported SUT's performance can be extrapolated to any production system of a similar scale and applications. Also, LDBC-SNB will give guidelines for a full disclosure report. The required information will ease the comparison of results between different systems, and ensure that these can be reproduced given the appropriate documentation and products. Last but not least, LDBC-SNB will require the results to be audited to be official. LDBC-SNB will define the necessary steps to follow during the audit process.

Finally, detailed instructions to generate the required datasets and to run Interactive Workload of the benchmark, are described in Chapter 3

1.4 Participation of Industry and Academia

The list of institutions that take part in the definition and development of LDBC-SNB is formed by relevant actors from both the industry and academia in the field of linked data management. All the participants have contributed with their experience and expertise in the field, making a credible and relevant benchmark that meets all the desired needs. The list of participants is the following:

- FOUNDATION FOR RESEARCH AND TECHNOLOGY HELLAS
- NEO4J
- ONTOTEX
- OPENLINK
- TECHNISCHE UNIVERSITAET MUENCHEN
- UNIVERSITAET INNSBRUCK
- UNIVERSITAT POLITECNICA DE CATALUNYA
- VRIJE UNIVERSITEIT AMSTERDAM

Besides the aforementioned institutions, during the development of the benchmark several meetings with the technical and users community have been conducted, receiving an invaluable feedback that has contributed to the whole development of the benchmark in every of its aspects.

2 Formal Definition

2.1 Requirements

LDBC-SNB is designed to be flexible and to have an affordable entry point. From small single node and in memory systems to large distributed multi-node clusters have its own place in LDBC-SNB. Therefore, the requirements to fulfill to execute LDBC-SNB are limited to pure software requirements to be able to run the tools. All the software provided by LDBC-SNB have been developed and tested under Linux, and use the following technologies:

- Java Development Kit 1.6 or newer.
- Hadoop 1.2.1
- Python 2.7

LDBC-SNB does not impose the usage of any specific type of system, as it targets systems of different nature and characteristics, from graph databases, graph processing frameworks and RDF systems, to traditional relation database management systems. Consequently, any language or API capable of expressing the proposed queries can be used. Similarly, data can be stored in the most convenient manner the test sponsor may decide, as long as it conforms with the execution rules. Finally, in order to have an official benchmark execution, the results will have to be audited and all the required information disclosed.

2.2 Data

This section introduces the data used by LDBC-SNB. This includes the different data types, the data schema, how it is generated and the different scale factors.

2.2.1 Data Types

Table 2.1 describes the different types used in the whole benchmark.

Type	Description
ID	integer type with 64-bit precision. All IDs within a single entity, are unique
32-bit Integer	integer type with 32-bit precision
64-bit Integer	integer type with 64-bit precision
String	variable length text of size 40
Text	variable length text of size 2000
Date	date with a precision of a day
DateTime	date with a precision of a second

Table 2.1: Description of the data types.

2.2.2 Data Schema

Figure 2.1 shows the data schema in UML. The schema defines the structure of the data used in the benchmark in terms of entities and their relations. Data represents a snapshot of the activity of a social network during a period of time. Data includes entities such as Persons, Organizations, and Places. The schema also models the way persons interact, by means of the friendship relations established with other persons, and the sharing of content such as messages (both textual and images), replies to messages and likes to messages. People form groups to talk about specific topics, which are represented as tags.

LDBC-SNB has been designed to be flexible and to target systems of different nature and characteristics. As such, it does not force any particular internal representation of the schema. The DBGEN described in Section 3.1 supports multiple output data formats to fit the needs of different types of systems, including RDF, relational DBMS and graph DBMS.

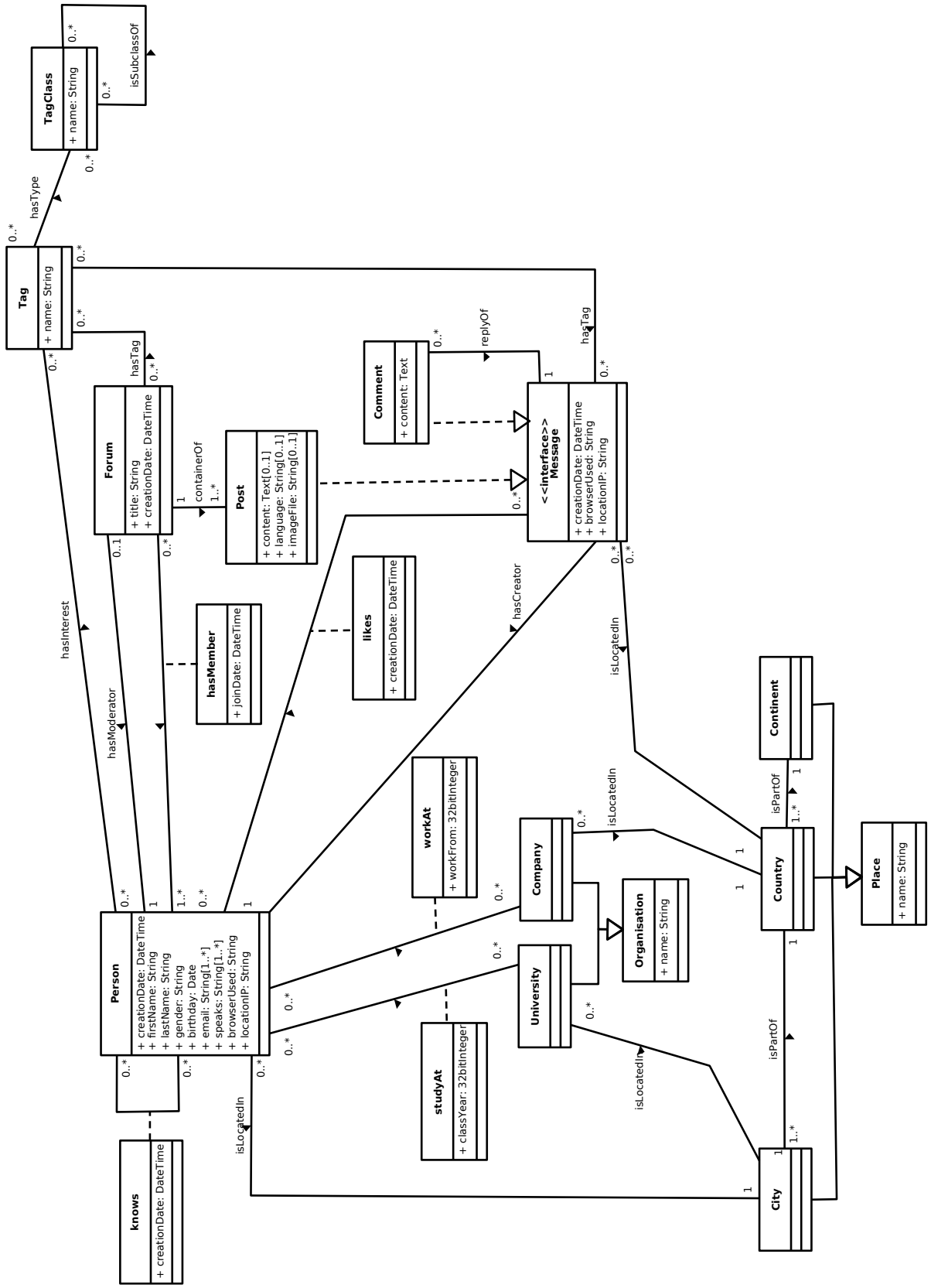


Figure 2.1: The LDBC-SNB data schema

The schema specifies different entities, their attributes and their relations. All of them are described in the following sections.

Entities

City: Is a sub-class of a Place, and represents a city of the real world. City entities are used to specify where persons live, as well as where universities operate.

Comment: Is a sub-class of a Message, and represents a comment made by a person to an existing message (either a Post or a Comment). Table 2.2 shows the attributes of Comment entity.

Attribute	Type	Description
content	Text	Contains the textual content of the comment.
length	32-bit Integer	The length of the comment.

Table 2.2: Attributes of Comment entity.

Continent: Is a sub-class of an Organization, and represents a company where persons work.

Country: Is a sub-class of a Place, and represents a continent of the real world.

Forum: Is forum represents a meeting point where people post messages. Forums are characterized by the topics (represented as tags) people in the forum are talking about. Although from the schema's perspective all the forums are equally considered, there exists three different types of forums: persons' personal walls, image albums, and groups. They are distinguished by their titles. Table 2.3 shows the attributes of Forum entity.

Attribute	Type	Description
id	ID	The identifier of the forum.
title	String	The title of the forum.
creationDate	DateTime	The date the forum was created

Table 2.3: Attributes of Forum entity.

Message: Is an abstract entity that represents a message created by a person. Table 2.4 shows the attributes of Message abstract entity.

Attribute	Type	Description
id	ID	The identifier of the message.
browserUsed	String	The browser used by the Person to create the message.
creationDate	DateTime	The date the message was created.
locationIP	String	The IP of the location from which the message was created.

Table 2.4: Attributes of Message interface.

Organization: Represents an institution of the real world. Table 2.5 shows the attributes of Organization entity.

Attribute	Type	Description
id	ID	The identifier of the organization.
name	String	The name of the organization.

Table 2.5: Attributes of Organization entity.

Person: Represents the avatar a real world person creates when he/she joins the network, and contains several information about the person as well as network related information. Table 2.6 shows the attributes of Person entity.

Attribute	Type	Description
id	ID	The identifier of the person.
firstName	String	The first name of the person.
lastName	String	The last name of the person.
gender	String	The gender of the person.
birthDay	Date	The birthday of the person .
email	String[1..*]	The set of emails the person has.
speaks	String[1..*]	The set of languages the person speaks.
browserUser	String	The browser used by the person when he/she registered to the social network.
locationIp	String	The IP of the location from which the person was registered to the social network.
creationDate	DateTime	The date the person joined the social network.

Table 2.6: Attributes of Person entity.

Place: Represents a place in the world. Table 2.7 shows the attributes of Place entity.

Attribute	Type	Description
id	ID	The identifier of the place.
name	String	The name of the place.

Table 2.7: Attributes of Place entity.

Post: Is a sub-class of Message, that is posted in a forum. Posts are created by persons into the forums where they belong. Posts contain either a textual content, or an image, but never both. Table 2.8 shows the attributes of Post entity.

Attribute	Type	Description
content	Text[0..1]	Contains the textual content of the post.
length	32-bit Integer	The length of the post.
language	String[0..1]	The language of the post.
imageFile	String[0..1]	The image file of the post..

Table 2.8: Attributes of Post entity.

Tag: Represents a topic or a concept. Tags are used to specify the topics of forums and posts, as well as the topics a person is interested in. Table 2.9 shows the attributes of Tag entity.

Attribute	Type	Description
id	ID	The identifier of the tag.
name	String	The name of the tag.

Table 2.9: Attributes of Tag entity.

TagClass: Represents a class or a category used to build a hierarchy of tags. Table 2.10 shows the attributes of TagClass entity.

Attribute	Type	Description
id	ID	The identifier of the tagclass.
name	String	The name of the tagclass.

Table 2.10: Attributes of TagClass entity.

University: Is a sub-class of Organization, and represents an institution where persons study.

Relations

Relations connect entities of different types. Entities are defined by their ”id” attribute.

Name	Tail	Head	Type	Description
containerOf	Forum[1]	Post[1..*]	D	A Forum and a Post contained in it
hasCreator	Message[0..*]	Person[1]	D	A Message and its creator (Person)
hasInterest	Person[0..*]	Tag[0..*]	D	A Person and a Tag representing a topic the person is interested in
hasMember	Forum[0..*]	Person[1..*]	D	A Forum and a member (Person) of the forum • **Attribute**: joinDate • **Type:** DateTime • **Description:** The Date the person joined the forum
hasModerator	Forum[0..*]	Person[1]	D	A Forum and his moderator (Person)
hasTag	Message[0..*]	Tag[0..*]	D	A Message and a Tag representing the message’s topic
hasTag	Forum[0..*]	Tag[0..*]	D	A Forum and a Tag representing the forum’s topic
hasType	Tag[0..*]	TagClass[0..*]	D	A Tag and a TagClass the tag belongs to
isLocatedIn	Company[0..*]	Country[1]	D	A Company and its home Country
isLocatedIn	Message[0..*]	Country[1]	D	A Message and the Country from which it was issued
isLocatedIn	Person[0..*]	City[1]	D	A Person and its home City
isLocatedIn	University[0..*]	City[1]	D	A University and the City where the university is
isPartOf	City[1..*]	Country[1]	D	A City and the Country it is part of
isPartOf	Country[1..*]	Continent[1]	D	A Country and the Continent it is part of
isSubclassOf	TagClass[0..*]	TagClass[0..*]	D	A TagClass its parent TagClass
knows	Person[0..*]	Person[0..*]	U	Two Persons that know each other • **Attribute**: creationDate • **Type:** DateTime • **Description:** The date the knows relation was established

likes	Person[0..*]	Message[0..*]	D	A Person that likes a Message • **Attribute**: creationDate • **Type:** DateTime • **Description:** The date the like was issued
replyOf	Comment[0..*]	Message[1]	D	A Comment and the Message it replies
studyAt	Person[0..*]	University[0..*]	D	A Person and a University it has studied • **Attribute**: classYear • **Type:** 32-bit Integer • **Description:** The year the person graduated.
workAt	Person[0..*]	Company[0..*]	D	A Person and a Company it works • **Attribute**: workFrom • **Type:** 32-bit Integer • **Description:** The year the person started to work at that company

Table 2.11: Description of the data relations.

2.2.3 Data Generation

LDBC-SNB provides DBGEN (Data Base Generator), which produces synthetic datasets following the schema described above. As described above, data produced mimics a social network's activity during a period of time. Three parameters determine the generated data: the number of persons, the number of years simulated, and the starting year of simulation. DBGEN is defined by the following characteristics:

- **Realism.** Data generated by DBGEN mimics the characteristics of those found in a real social network. In DBGEN, output attributes, cardinalities, correlations and distributions have been finely tuned to reproduce a real social network in each of its aspects On the one hand, it is aware of the data and link distributions found in a real social network such as Facebook. On the other hand, it uses real data from DBPedia, such as property dictionaries, which are used to ensure that attribute values are realistic and correlated.
- **Scalability.** Since LDBC-SNB targets systems of different scales and budgets, DBGEN is capable of generating datasets of different sizes, from a few Gigabytes to Terabytes. DBGEN is implemented following the MapReduce parallel paradigm, allowing the generation of small datasets in single node machines, as well as large datasets on commodity clusters.
- **Determinism.** DBGEN is deterministic regardless of the number of cores/machines used to produce the data. This important feature guarantees that all Test Sponsors will face the same dataset, thus, making the comparisons between different systems fair and the benchmarksâĂŹ results reproducible.
- **Usability.** LDBC-SNB is designed to have an affordable entry point. As such, DBGEN's design is severely influenced by this philosophy, and therefore it is designed to be as easy to use as possible.

Property Dictionaries and Resource Files

DBGEN uses a set of property dictionaries and other resource files with data extracted from DBpedia. Conceptually, DBGEN generates attribute's values following a property dictionary model that is defined by

- a dictionary D
- a ranking function R
- a probability function F

Dictionary D is a fixed set of values. The ranking function R is a bijection that assigns to each value in a dictionary a unique rank between 1 and $|D|$. The probability density function F specifies how the data generator chooses values from dictionary D using the rank for each term in the dictionary. The idea to have a separate ranking and probability function is motivated by the need of generating correlated values: in particular, the ranking function is typically parameterized by some parameters: different parameter values result in different rankings. For example, in the case of a dictionary of property firstName, the popularity of first names, might depend on the gender, country and birthDate properties. Thus, the fact that the popularity of first names in different countries and times is different, is reflected by the different ranks produced by function R over the full dictionary of names. DBGEN uses a dictionary for each literal property, as well as ranking functions for all literal properties. These are materialized in a set of resource files, which are described in Table 2.12.

Resource Name	**Description**
Browsers	Contains a list of web browsers and their probability to be used. It is used to set the browsers used by the users.
Cities by Country	Contains a list of cites and the country they belong. It is used to assign cities to users and universities.
Companies by Country	Contains the set of companies per country. It is used to set the countries where companies operate.
Countries	Contains a list of countries and their populations. It is used to obtain the amount of people generated for each country.
Emails	Contains the set of email providers. It is used to generate the email accounts of persons.
IP Zones	Contains the set of IP ranges assigned to each country. It is used to assign the IP addresses to users.
Languages by Country	Contains the set of languages spoken in each country. It is used to set the languages spoken by each user.
Name by Country	Contains the set of names and the probability to appear in each country. It is used to assign names to persons, correlated with their countries.
Popular places by Country	Contains the set of popular places per country. These are used to set where images attached to posts are taken from.
Surnames' by Country	Contains the set of surnames and the probability to appear in each country. It is used to assign surnames to persons, correlated with their countries.
Tags by Country	Contains a set of tags and their probability to appear in each country. It is used to assign the interests to persons and forums.
Tag Classes	Contains, for each tag, the classes it belongs to.
Tag Hierarchies	Contains, for each tagClass, their parent tagClass.
Tag Matrix	Contains, for each tag, the correlation probability with the other tags. It is used enrich the tags associated to messages.
Tag Text	Contains, for each tag, a text. This is used to generate the text for messages.
Universities by City	Contains the set of universities per city. It is used to set the cities where universities operate.

Table 2.12: Property dictionaries and resource files

Graph Generation

Figure 2.2 conceptually depicts the full data generation process. The first step loads all the dictionaries and resource files, and initializes the DBGEN parameters. Second, it generates all the Persons in the graph, and the minimum necessary information to operate. Part of these information are the interests of the persons, and the number of knows relationships of every person, which is guided by a degree distribution function similar to that found in Facebook [4].

The next three steps are devoted to the creation of knows relationships. An important aspect of real social networks, is the fact that similar persons (with similar interests and behaviors) tend to be connected. This is known as the Homophily principle [3], and implies the presence of a larger amount of triangles than that expected in a random network. In order to reproduce this characteristic, DBGEN generates the edges by means of correlation dimensions. Given a person, the probability to be connected to another person is typically skewed with respect to some similarity between the persons. That is, for a person n and for a small set of persons that are somehow similar to it, there is a high connectivity probability, whereas for most other persons, this probability is quite low. This knowledge is exploited by DBGEN to reproduce correlations.

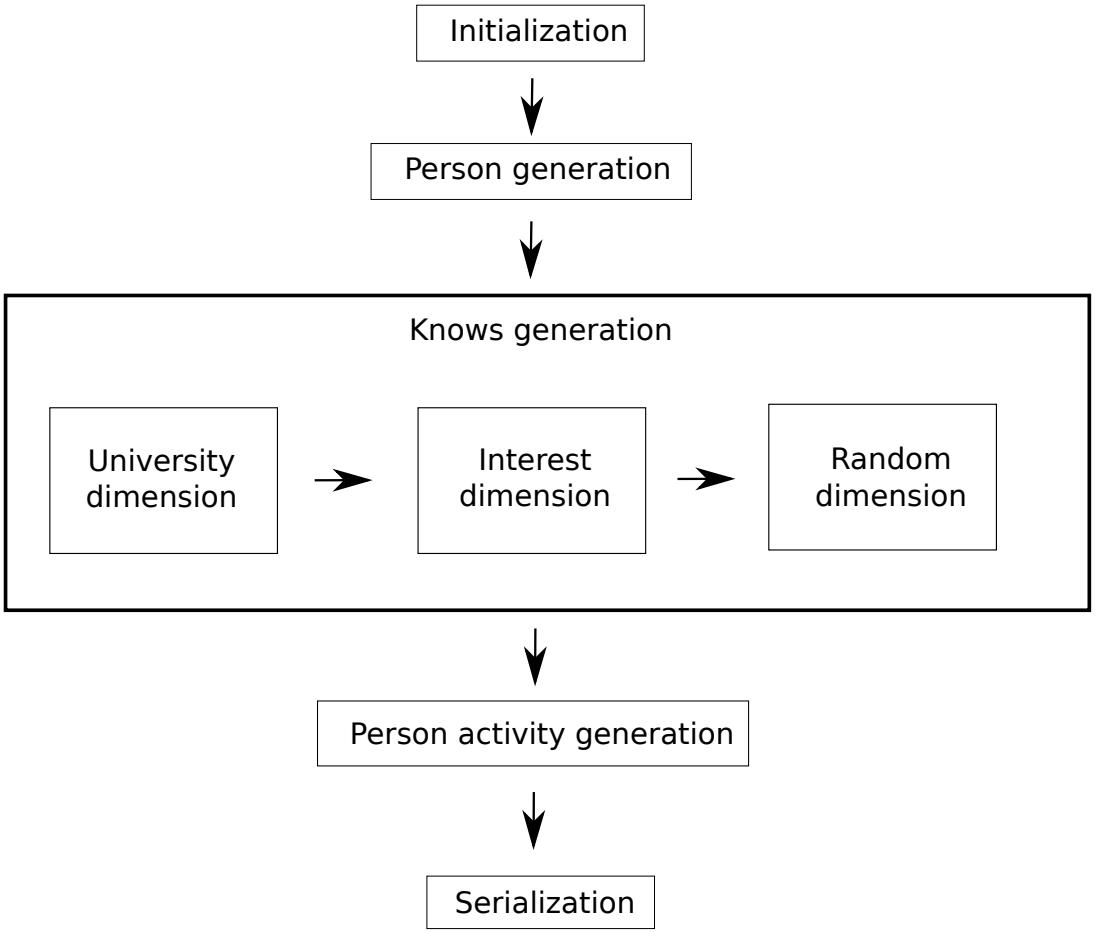


Figure 2.2: The DBGEN generation process.

Given a similarity function $M(x) : n[0, \infty]$ that gives a score to a person, with the characteristic that two similar persons will have similar scores, we can sort all the persons by function M and compare a person n against only the W neighboring persons in the sorted array. The consequence of this approach is that similar persons are grouped together, and the larger the distance between two persons indicates a monotonic increase in their similarity difference. In order to choose the persons to connect, DBGEN uses a geometric probability

distribution that provides a probability for picking persons to connect, that are between 1 and W positions apart in the similarity ranking.

Similarity functions and probability distribution functions over ranked distance drive what kind of persons will be connected with an edge, not how many. As stated above, the number of friends of a person is determined by a Facebook-like distribution. The edges that will be connected to a person n, are selected by randomly picking the required number of edges according to the correlated probability distributions as discussed before. In the case that multiple correlations exist, another probability function is used to divide the intended number of edges between the various correlation dimensions. In DBGEN, three correlated dimensions are chosen: the first one depends on where the person studied and when, and the second correlation dimension depends on the interests of the person, and the third one is random (to reproduce the random noise present in real data). Thus, DBGEN has a Facebook-like distributed node degree, and a predictable (but not fixed) average split between the reasons for creating edges.

In the next step, person's activity, in the form of forums, posts and comments is created. DBGEN reproduces the fact that people with a larger number of friends have a higher activity, and hence post more photos and comments to a larger number of posts. Another important characteristic of real users' activity in social network, are time correlations. Usually, users' posts creation in a social network is driven by real world events. For instance, one may thing about an important event such as the elections in a country, or a natural disaster. Around the time these events occur, network activity about these events' topics sees an increase in volume. DBGEN reproduces these characteristics with the simulation of what we name as flashmob events. Several events are generated randomly at the beginning of the generation process, which are assigned a random tag, and are given a time and an intensity which represents the repercussion of the event in the real world. When persons' posts are created, some of them are classified as flashmob posts, and their topics and dates are assigned based on the generated flashmob events. The volume of activity around this events is modeled following a model similar to that described in [2]. Furthermore, in order to reproduce the more uniform every day's user activity, DBGEN also generates post uniformly distributed along all the simulated time.

Finally, in the last step the data is serialized into the output files.

Implementation Details

DBGEN is implemented using the MapReduce parallel paradigm. In MapReduce, a Map function runs on different parts of the input data, in parallel and on many node clusters. This function processes the input data and produces for each result a key. Reduce functions then obtain this data and Reducers run in parallel on many cluster nodes. The produced key simply determines the Reducer to which the results are sent. The use of the MapReduce paradigm allows the generator to scale considerably, allowing the generation of huge datasets by using clusters of machines.

In the case of DBGEN, the overall process is divided into three MapReduce jobs. In the first job, each mapper generates a subset of the persons of the graph. A key is assigned to each person using one of the similarity functions described above. Then, reducers receive the the key-value pairs sorted by the key, generate the knows relations following the described windowing process, and assign to each person a new key based on another similarity function, for the next MapReduce pass. This process can be successively repeated for additional correlation dimension. Finally, the last reducer generates the remaining information such as forums, posts and comments.

Data Output

DBGEN is built to split the simulated social network into two parts: the static part and the update stream part. The static part contains the data that will be bulk loaded by the Test Sponsor's system and is formatted in one of the supported formats: CSV, CSV_MERGE_FOREIGN and TTL. In addition to the network data, a stream of reads is also produced, which is used by the test driver to issue the queries. A detailed description of each supported format and the generated files is described in Section 3.1.2. For a description of how the read stream is generated, please refer to Section 2.3.1. The update streams part contains update events to the network, consisting in insertions of data, and is also used by the test driver to issue updates.

What percentage of the network is output as static, and what percentage is output as updates, can be configured and depends on the needs of the workload. To compute what data goes to each of the parts, a point or threshold in the simulated time line is computed. All entities that fall before the threshold, go to the static part. All entities falling after the threshold, are output as update streams. Consequently, if a relation contains one of the entities falling into the update stream, then the relation is also output as an update.

As currently the only supported workload is the Interactive Workload with only lookups, DBGEN is configured to output the entire 100% of the network as static by default.

2.2.4 Scale Factors

LDBC-SNB defines a set of scale factors (SFs), targeting systems of different sizes and budgets. SFs are computed based on the ASCII size in Gigabytes of the generated output files using the CSV serializer. For example, SF 1 weights roughly 1GB in CSV format, SF 3 weights roughly 3GB and so on and so forth. The proposed SFs are the following: 1, 3, 10, 30, 100, 300, 1000. The Test Sponsor may select the SF that better fits their needs, by properly configuring the DBGEN, as described in Section 3.1.

The size of the resulting dataset, is mainly affected by the following configuration parameters: the number of persons and the number of years simulated. Different SFs are computed by scaling the number of Persons in the network, while fixing the number of years simulated. Table 2.13 shows the parameters used in each of the SFs.

Scale Factor	1	3	10	30	100	300	1000
# of Persons	11K	27K	73K	182K	499K	1.25M	3.6M
# of Years	3	3	3	3	3	3	3
Start Year	2010	2010	2010	2010	2010	2010	2010

Table 2.13: Parameters of each scale factor.

For example, SF 100 consists of the activity of a social network of 182K users during a period of three years, starting from 2010. In Appendix B.1, we show the statistics of each of the proposed SFs in detail, including distributions for some of the relations.

2.3 Workloads

2.3.1 Interactive Workload

Choke Points

The design of the interactive workload queries has been conceived around two main aspects: realism and technological relevance. While realism has been assessed by looking at existing social networks and thinking about what interesting functionalities a user might desire from them, technological relevance has been achieved by identifying a set of choke points queries should stress. These choke points capture those critical operations, techniques or technologies that could significatively affect the performance of the queries. The choke points can be summarized in the following list:

- **Aggregation Performance.**

  The queries generally have a top k order by and often a group by in addition to this. These offer multiple optimization opportunities. The queries also often have distinct operators, i.e. distinct friends within two steps. Collectively these are all set operations that may be implemented with some combination of hash and sorting, possibly exploiting ordering in the data itself. The aggregates are not limited to counts and sums. For example string concatenation occurs as an aggregate, testing possible user defined aggregate support. There is a wide range of cardinalities in grouping, from low, e.g. country, to high, e.g. post.

- **Join Performance.**

  Each graph traversal step is in principle a join. The join patterns are diverse, exercising both index and hash based operators. Queries are designed so as to reward judicious use of hash join by having patterns starting with one entity, fetching many related entities and then testing how these are related to a third entity, e.g. posts of a user with a tag of a given type.

- **Data Access Locality.**

  Graph problems are notoriously non-local. However, when queries touch any non-trivial fraction of a dataset, locality will emerge and can be exploited, for example by vectored index access or by ordering data so that that a merge join is possible.

- **Expression Calculation.**

  Queries often have expressions, including conditional expressions. This provides opportunities for vectoring and tests efficient management of intermediate results.

- **Correlated Subqueries.**

  The workload has many correlated subqueries, for example constructts like x within two steps but not in one step, which would typically be a correlated subquery with not exists. There are also scalar subqueries with aggregation, for example returning the count of posts satisfying a certain criteria.

- **Parallelism and Concurrency.**

  All queries offer opportunities for parallelism. This tests a wide range of constructs, for example partitioned parallel variants of group by and distinct. An interactive workload will typically avoid trivially parallelizable table scans. Thus the opportunities that exist must be captured by index based, navigational query plans. The choice of whether to parallelize or not is often left to run time and will have to depend on the actual data seen in the execution, as starting a parallel thread with too little to do is counter-productive.

- **Graph Specifics.**

  Graph problems are generally characterized by transitive properties and the fact that neighboring vertices often have a large overlap in their environments. This makes cardinality estimation harder. For example, a query optimizer needs to recognize whether a relationship has a tree or graph shape in order to make correct cardinality estimations. Further, there are problems aggregating properties over a set of consecutive edges. The workload contains business questions dealing with paths and aggregates across paths, as well as the easier case of determining a membership in a hierarchy with a transitive part-of relation.

Query Description Format

Queries are described in natural language using a well-defined structure that consists of three sections: *description*, a concise textual description of the query; *parameters*, a list of input parameters and their types; and *results*, a list of expected results and their types. The syntax used in *parameters* and *results* sections is as follows:

- **Entity**: entity type in the dataset.
  One word, possibly constructed by appending multiple words together, starting with uppercase character and following the camel case notation, e.g. `TagClass` represents an entity of type "TagClass".
- **Relationship**: relationship type in the dataset.
  One word, possibly constructed by appending multiple words together, starting with lowercase character and following the camel case notation, and surrounded by arrow to communicate direction, e.g. `-worksAt->` represents a directed relationship of type "worksAt".
- **Attribute**: attribute of an entity or relationship in the dataset.
  One word, possibly constructed by appending multiple words together, starting with lowercase character and following the camel case notation, and prefixed by a "." to dereference the entity/relationship, e.g. `Person.firstName` refers to "firstName" attribute on the "Person" entity, and `-studyAt->.classYear` refers to "classYear" attribute on the "studyAt" relationship.

- **Unordered Set**: an unordered collection of distinct elements.
  Surrounded by { and } braces, with the element type between them, e.g. `{String}` refers to a set of strings.
- **Ordered List**: an unordered collection where duplicate elements are allowed.
  Surrounded by [and] braces, with the element type between them, e.g. `[String]` refers to a list of strings.
- **Ordered Tuple**: a fixed length, fixed order list of elements, where elements at each position of the tuple have predefined, possibly different, types.
  Surrounded by < and > braces, with the element types between them in a specific order e.g. `<String, Boolean>` refers to a 2-tuple containing a string value in the first element and a boolean value in the second, and `[<String, Boolean>]` is an ordered list of those 2-tuples.

Query Descriptions

1. Friends with certain name
   - **Description:** Given a start Person, find up to 20 Persons with a given first name that the start Person is connected to (excluding start Person) by at most 3 steps via Knows relationships. Return Persons, including summaries of the Persons' workplaces and places of study. Sort results by their distance from the start Person, for Persons within the same distance sort by their last name, and for Persons with same last name by their identifier
   - **Parameters:**

Person.id	ID
Person.firstName	String

   - **Results:**

Person.id	ID
Person.lastName	String
Person.birthday	Date
Person.creationDate	DateTime
Person.gender	String
Person.browserUsed	String
Person.locationIP	String
{Person.emails}	{String}
{Person.language}	{String}
Person-isLocatedIn->Location.name	String
{Person-workAt->Company.name, Person-workAt->.workFrom, Person-workAt->Company-isLocatedIn->City.name}	{<String, 32-bit Integer, String>}
{Person-workAt->Company.name, Person-workAt->.workFrom, Person-workAt->Company-isLocatedIn->City.name}	{<String, 32-bit Integer, String>}

2. Recent posts and comments by your friends
   - **Description:** 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.
   - **Parameters:**

Person.id	ID
date	DateTime

   - **Results:**

Person.id	ID
Person.firstName	String
Person.lastName	String
Post.id/Comment.id	ID
Post.content/Post.imageFile/Comment.content	String
Post.creationDate/Comment.creationDate	DateTime

3. Friends and friends of friends that have been to countries X and Y

- **Description:** Given a start Person, find Persons that are their friends and friends of friends (excluding start Person) that have made Posts/Comments in the given Countries X and Y within a given period. Only Persons that are foreign to Countries X and Y are considered, that is Persons whose Location is not Country X or Country Y. Return top 20 Persons, and their Post/Comment counts. Sort results descending by total number of Posts or Comments, and then ascending by Person identifier.
- **Parameters:**

Person.id	ID	
CountryX.name	String	
CountryY.name	String	
startDate	DateTime	// beginning of requested period
duration	32-bit Integer	// duration of requested period, in days

- **Results:**

Person.id	ID	
Person.firstName	String	
Person.lastName	String	
countx	32-bit Integer	// number of Posts/Comments from Country X made by Person within the given time
county	32-bit Integer	// number of Posts/Comments from Country Y made by Person within the given time
count	32-bit Integer	// countx + county

4. New topics

- **Description:** Given a start Person, find Tags that are attached to Posts that were created by that Person's friends. Only include Tags that were attached to Posts created within a given time interval, and that were never attached to Posts created before this interval. Return top 10 Tags, and the count of Posts, which were created within the given time interval, that this Tag was attached to. Sort results descending by Post count, and then ascending by Tag name.
- **Parameters:**

Person.id	ID	
startDate	DateTime	
duration	32-bit Integer	// number of days

- **Results:**

Tag.name	String	
count	32-bit Integer	// number of Posts made within the given time interval that have this Tag

5. New groups

- **Description:** Given a start Person, find the Forums which that Person's friends and friends of friends (excluding start Person) became Members of after a given date. Return top 20 Forums, and the number of Posts in each Forum that was Created by any of these Persons. Sort results descending by the count of Posts, and then ascending by Forum name.
- **Parameters:**

Person.id	ID
date	DateTime

- **Results:**

Forum.title	String	
count	32-bit Integer	// number of Posts made in Forum that were created by friends

6. Tag co-occurrence

- **Description:** Given a start Person and some Tag, find the other Tags that occur together with this Tag on Posts that were created by start Person's friends and friends of friends (excluding start Person). Return top 10 Tags, and the count of Posts that were created by these Persons, which contain this Tag. Sort results descending by count, and then ascending by Tag name.
- **Parameters:**

Person.id	ID
Tag.name	String

- **Results:**

Tag.name	String	
count	32-bit Integer	// number of Posts that were created by friends and friends of friends, which contain this Tag

7. Recent likes

- **Description:** Given a start Person, find (most recent) Likes on any of start Person's Posts/Comments. Return top 20 Persons that Liked your Post/Comment, the Post they liked, the Like, and the latency between creation of Post/Comment and Like. Additionally, return a flag indicating whether the liker is a friend of start Person. Sort results descending by creation time of Like, and then ascending by Person identifier of liker.
- **Parameters:**

Person.id	64-bit Integer

- **Results:**

Person.id	ID	
Person.firstName	String	
Person.lastName	String	
Like.creationDate	DateTime	
Post.id/Comment.id	ID	
Post.content/Post.imageFile/Post.content	String	
latency	32-bit Integer	// duration between creation of Post/Comment and Like, in minutes
isNew	Boolean	// false if liker Person is friend of start Person, true otherwise

8. Recent replies

- **Description:** Given a start Person, find (most recent) Comments that are Replies to Posts/Comments of the start Person. Return the top 20 reply Comments, and the Person that created each reply Comment. Sort results descending by creation date of reply Comment, and then ascending by identifier of reply Comment.
- **Parameters:**

Person.id	ID

- **Results:**

Person.id	ID
Person.firstName	String
Person.lastName	String
Comment.creationDate	DateTime
Comment.id	ID
Comment.content	String

9. Recent posts and comments by friends or friends of friends

- **Description:** Given a start Person, find the (most recent) Posts/Comments created by that Person's friends or friends of friends (excluding start Person). Only consider the Posts/Comments created before a given date (excluding that date). Return the top 20 Posts/Comments, and the Person that created each of those Posts/-Comments. Sort results descending by creation date of Post/Comment, and then ascending by Post/Comment identifier.
- **Parameters:**

Person.id	ID
date	DateTime

- **Results:**

Person.id	ID
Person.firstName	String
Person.lastName	String
Post.id/Comment.id	ID
Post.content/Post.imageFile/Comment.content	String
Post.creationDate/Comment.creationDate	DateTime

10. Friend recommendation

- **Description:** Given a start Person, find that Person's friends of friends (excluding start Person, and immediate friends), who were born on or after the 21st of a given month (in any year) and before the 22nd of the following month. Calculate the similarity between each of these Persons and start Person, where similarity for any Person is defined as follows:
  - common = number of Posts created by that Person, such that the Post has a Tag that start Person is Interested in
  - uncommon = number of Posts created by that Person, such that the Post has no Tag that start Person is Interested in
  - similarity = common - uncommon

  Return top 10 Persons, their Location, and their similarity score. Sort results descending by similarity score, and then ascending by Person identifier
- **Parameters:**

Person.id	ID	
month1	32-bit Integer	// between 1-12
month2	32-bit Integer	// month1 + 1, but 12 + 1 = 1

- **Results:**

Person.id	ID
Person.firstName	String
Person.lastName	String
Person.gender	String
Person-isLocatedIn->Location.name	Sting
similarity	32-bit Integer

11. Job referral

- **Description:** Given a start Person, find that Person's friends and friends of friends (excluding start Person) who started Working in some Company in a given Country, before a given date (year). Return top 10 Persons, the Company they worked at, and the year they started working at that Company. Sort results ascending by the start date, then ascending by Person identifier, and lastly by Organization name
- **Parameters:**

Person.id	ID
Country.name	String
year	32-bit Integer

- **Results:**

Person.id	ID
Person.firstName	String
Person.lastName	String
Person-worksAt->.worksFrom	32-bit Integer
Person-worksAt->Organization.name	String

12. Expert search

- **Description:** Given a start Person, find the Comments that this Person's friends made in reply to Posts. Only consider Posts with a Tag in a given TagClass or in a descendent of that TagClass. Count the number of these reply Comments, and collect the Tags that were attached to the Posts they replied to. Return top 20 Persons, the reply count, and the collection of Tags. Sort results descending by Comment count, and then ascending by Person identifier.
- **Parameters:**

Person.id	ID
TagClass.id	ID

- **Results:**

Person.id	ID	
Person.firstName	String	
Person.lastName	String	
Tag.name	String	
count	32-bit Integer	// number of reply Comments

13. Single shortest path

    - **Description:** Given two Persons, find the shortest path between these two Persons in the subgraph induced by the Knows relationships. Return the length of this path.
    - **Parameters:**
      Person.id ID // person 1
      Person.id ID // person 2
    - **Results:**
      length 32-bit Integer

14. Weighted paths

    - **Description:** Given two Persons, find all weighted paths of the shortest length between these two Persons in the subgraph induced by the Knows relationship. The nodes in the path are Persons. Weight of a path is sum of weights between every pair of consecutive Person nodes in the path. The weight for a pair of Persons is calculated such that every reply (by one of the Persons) to a Post (by the other Person) contributes 1.0, and every reply (by ones of the Persons) to a Comment (by the other Person) contributes 0.5. Return all the paths with shortest length, and their weights. Sort results descending by path weight.
    - **Parameters:**
      Person.id ID // person 1
      Person.id ID // person 2
    - **Results:**
      [Person.id] [ID] // Identifiers representing an ordered sequence of the Persons in the path
      weight 64-bit Float

Substitution parameters

Together with the dataset, DBGEN produces a set of parameters per query type. Parameter generation is designed in such a way that for each query type, all of the generated parameters yield similar runtime behaviour of that query. Specifically, the amount of data that the query touches is roughly the same for every parameter binding, assuming that the query optimizer figures out a reasonable execution plan for the query. This is done to avoid bindings that cause unexpectedly long or short runtimes of queries, or even result in a completely different optimal execution plan. Such effects could arise due to the data skew and correlations between values in the generated dataset.

Parameter bindings are stored in the `substitution_parameters` folder inside the data generator directory. Each query gets its bindings in a separate file. Every line of a parameter file is a JSON-formatted collection of key-value pairs (name of the parameter and its value). For example, the Query 1 parameter bindings are stored in file `query_1_param.txt`, and one of its lines may look like this:

{"PersonID" : 1, "Name" : "Lei", "PersonURI" : "http://www.ldbc.eu/ldbc_socialnet/1.0/data/pers1"}

Depending on implementation, the SUT may refer to persons either by IDs (relational and graph databases) or URIs (RDF systems), so we provide both values for the Person parameter.

Load Definition

In addition to specifying query types and parameters, the Interactive Workload definition also includes the target load. That is, the frequency/throughput at which queries will be executed.

Rather than simply measuring the maximum throughput an SUT can achieve given some query mix, the Interactive Workload explicitly defines the rate that queries will be sent to the SUT, and query response times (latencies) are then measured for these queries. This philosophy is more in line with the *interactive* nature of the benchmark. Further, specifying a particular query rate allows the benchmark to measure how an SUT performs, not only with a given set of query types, but also under under a given load.

More specifically, the Interactive Workload includes two distinct groups of queries, reads and writes. Although writes will be disabled for the initial benchmark release, it is important to understand how reads and writes relate to one another, and to be aware of the difference between “simulation time” and “real time”.

Both groups (reads and writes) are loaded from "event stream" files that DBGEN produces, but there is one important distinction between them: the files for reads contain substitution parameters only, the files for writes contain both substitution parameters and timestamps that indicate the scheduled execution time of each operation. One of the reasons writes include timestamps is they are dependent on one another, that is, to ensure data consistency they must be executed in the correct order.

Further, due to the fact that write timestamps are created in the data generator, they map to what we refer to as "simulation time": the timestamps may be in the past or future, but are always between the start and end times of the generated dataset; the scheduled execution time of an "add post" operation, for example, will be identical to the `creationDate` attribute of that same `Post` record. An example entry in the write stream file is as follows:

$$1293907146704 \| ADD\_LIKE\_POST \| [31277, 42949, \text{“}2011-01-01T19:39:06Z''] \|$$

On the other hand, as the data generator provides no associated timestamps for the read queries these must be created in the workload driver. As such, the driver provides parameters to control how timestamps are generated. The duration (in "simulation time") between queries can be set, individually per query type. Using these settings the driver generates multiple streams of read queries, on per query type, then merges them, along with the write stream provided by the data generator, into one time-ordered stream of read and write operations.

Finally, to provide a means of increasing or decreasing the frequency/throughput of operations issued by the workload driver, the driver provides a "time compression" (see 3.2.1) mechanism allowing timestamps in the generated operation stream to be "squeezed" closer together or "stretched" further apart, thereby controlling the rate at which they are executed.

3 Implementation Instructions

3.1 Data generation

DBGEN makes uses Hadoop to implement the data generation. Detailed instructions to configure hadoop for running DBGEN can be found at the DBGEN project software repository[1].

3.1.1 DBGEN Configuration, Compilation and Execution

DBGEN is designed to be as easy to configure, compile and execute as possible. With this objective in mind, a *run.sh* script is provided, which handles all the compilation and execution processes. *run.sh* is found in the DBGEN root folder. DBGEN uses Apache Maven to download any required dependencies and compile the sources. *run.sh* needs to be configured with two variables pointing to the proper folders. The following is the list of variables to set:

- **HADOOP_HOME**: Points to your hadoop root folder.
- **LDBC_SOCIALNET_DBGEN_HOME**: Points to your DBGEN root folder.

Once these variables are properly set, by typing:

```
$ sh run.sh
```

DBGEN is compiled and executed, and a dataset with the default options is generated in the current folder.

A file *params.ini* is used to change the characteristics of the generated network, as well as to set other options. Table 3.1 summarizes the different available options and their default values:

Option	Default	Description
scaleFactor	1	The scale factor of the data to generate. Possible values are: 1, 3, 10, 30, 100, 300 and 1000
serializer	csv	The format of the output data. Options are: csv, csv_merge_foreign, ttl
compressed	false	Specifies to compress the output data in gzip.
outputDir	./	Specifies the folder to output the data.
numThreads	1	Sets the number of threads to use. Only works for pseudo-distributed mode

Table 3.1: Description of the data types.

An example of *params.ini* for scale factor 30, ttl serializer, 4 threads, compressed output and a custom output dir, should look like:

```
scaleFactor:30
serializer:ttl
compressed:true
outputDir:/home/user/output
numThreads:4
```

DBGEN outputs data into HDFS. The outputDir directory specified in *params.ini* file, will be automatically created in HDFS if it does not exist. If there is not an HDFS file system mounted, then data is output to your local file system.

[1] DBGEN repository: https://github.com/ldbc/ldbc_socialnet_bm

3.1.2 Serializers

LDBC-SNB supports three different output formats: TTL, CSV and CSV_MERGE_FOREIGN. Besides the serializers' specific files, other files are generated: updateStream.csv files, which contains the update queries and is used by the test driver to issue the workload, and the substitution parameters files described in Section 2.3.1.

TTL

This is the standard Turtle[2] format. DBGEN outputs two files: 0_ldbc_socialnet_static_dbp.ttl and 0_ldbc_socialnet.ttl.

CSV

This is a comma separated format. Each entity, relation and properties with a cardinality larger than one, are output in a separate file. Generated files are summarized at Table 3.2. Depending on the number of threads used for generating the dataset, the number of files varies, since there is a file generated per thread. The * in the file names indicates a number between 0 and $NumberOfThreads - 1$.

[2]Description of the Turtle RDF format http://www.w3.org/TR/turtle/

File	Content
comment_*.csv	id | creationDate | locationIP | browserUsed | content | length |
comment_hasCreator_person_*.csv	Comment.id | Person.id |
comment_isLocatedIn_place_*.csv	Comment.id | Place.id |
comment_replyOf_comment_*.csv	Comment.id | Comment.id |
comment_replyOf_post_*.csv	Comment.id | Post.id |
forum_*.csv	id | title | creationDate |
forum_containerOf_post_*.csv	Forum.id | Post.id |
forum_hasMember_person_*.csv	Forum.id | Person.id | joinDate |
forum_hasModerator_person_*.csv	Forum.id | Person.id |
forum_hasTag_tag_*.csv	Forum.id | Tag.id |
organization_*.csv	id(Long) | type("university", "company") | name | url |
organisation_isLocatedIn_place_*.csv	Organisation.id | Place.id |
person_*.csv	id | firstName | lastName | gender | birthday | creationDate | locationIP | browserUsed |
person_email_emailaddress_*.csv	Person.id | email |
person_hasInterest_tag_*.csv	Person.id | Tag.id |
person_isLocatedIn_place_*.csv	Person.id | Place.id |
person_knows_person_*.csv	Person.id | Person.id | creationDate |
person_likes_comment_*.csv	Person.id | Post.id | creationDate |
person_likes_post_*.csv	Person.id | Post.id | creationDate |
person_speaks_language_*.csv	Person.id | language |
person_studyAt_organisation_*.csv	Person.id | Organisation.id | classYear |
person_workAt_organisation_*.csv	Person.id | Organisation.id | workFrom |
place_*.csv	id | name | url | type("city", "country", "continent") |
place_isPartOf_place_*.csv	Place.id | Place.id |
post_*.csv	id | imageFile | creationDate | locationIP | browserUsed | language | content | length |
post_hasCreator_person_*.csv	Post.id | Person.id |
post_hasTag_tag_*.csv	Post.id | Tag.id |
post_isLocatedIn_place.csv	Post.id | Place.id |
tag_*.csv	id | name | url |
tag_hasType_tagclass_*.csv	Tag.id | TagClass.id |
tagclass_*.csv	id | name | url |
tagclass_isSubclassOf_tagclass_*.csv	TagClass.id|TagClass.id |

Table 3.2: Files output by CSV serializer

CSV_MERGE_FOREIGN

This is a comma separated format. It is similar to CSV, but those relations connecting two entities A and B, where an entity A has a cardinality of one, A is output as a column of entity B. Generated files are summarized at Table 3.3. Depending on the number of threads used for generating the dataset, the number of files varies, since there is a file generated per thread. The * in the file names indicates a number between 0 and

File	Content
comment_*.csv	id | creationDate | locationIP | browserUsed | content | length | creator | place | replyOfPost | replyOfComment |
forum_*.csv	id | title | creationDate | moderator |
forum_hasMember_person_*.csv	Forum.id | Person.id | joinDate |
forum_hasTag_tag_*.csv	Forum.id | Tag.id |
organization_*.csv	id | type("university", "company") | name | url |
organisation_isLocatedIn_place_*.csv	Organisation.id | Place.id |
person_*.csv	id | firstName | lastName | gender | birthday | creationDate | locationIP | browserUsed | place |
person_email_emailaddress_*.csv	Person.id | email |
person_hasInterest_tag_*.csv	Person.id(Long) | Tag.id |
person_knows_person_*.csv	Person.id | Person.id | creationDate |
person_likes_comment_*.csv	Person.id | Post.id | creationDate |
person_likes_post_*.csv	Person.id | Post.id | creationDate |
person_speaks_language_*.csv	Person.id | language |
person_studyAt_organisation_*.csv	Person.id | Organisation.id | classYear |
person_workAt_organisation_*.csv	Person.id | Organisation.id | workFrom |
place_*.csv	id | name | url | type("city", "country", "continent") |
place_isPartOf_place_*.csv	Place.id | Place.id |
post_*.csv	id | imageFile | creationDate | locationIP | browserUsed | language | content | length | creator | Forum.id | place |
post_hasTag_tag_*.csv	Post.id | Tag.id |
tag_*.csv	id | name | url |
tag_hasType_tagclass_*.csv	Tag.id | TagClass.id |
tagclass_*.csv	id | name | url |
tagclass_isSubclassOf_tagclass_*.csv	TagClass.id | TagClass.id |

Table 3.3: Files output by CSV_MERGE_FOREIGN serializer

3.2 Running the benchmark

Running a benchmark workload involves a number of steps, including data import, driver configuration and workload execution. The data import step involves loading the output of the data generator into the vendor database, deploying the database and, optionally, performing a warm-up phase. The other steps are explained in the following sections.

3.2.1 Driver Configuration

Before running a benchmark workload the workload driver [1] must be configured, this involves the following steps:

1. Programming: implement a vendor specific database connector (covered in section 3.2.1)
2. Configuration: configure the general driver properties (covered in section 3.2.1)
3. Configuration: configure the workload specific driver properties (covered in 3.2.1)

Vendor specific database connector

Vendors must provide the workload driver with implementations for a number of Java classes: `Db` and `OperationHandler`. More specifically, one implementation of `Db`, (optionally) one implementation of `DbConnectionState` (used to manage shared resources among queries, e.g. network connections), and one implementation of `OperationHandler` per query in the workload.

As a guide, to implement the necessary classes for a workload of two queries (`LdbcQuery1` and `LdbcQuery2`), see the code snippet below. For a more detailed explanation of how this is done refer to the workload driver documentation [1].

```
public class ExampleDbConnectionState extends DbConnectionState {
    private Client client;
    public ExampleDbConnectionState(String url){
        client = new Client(url);
    }

    public getClient(){
        return client;
    }
}

public class ExampleDb extends Db {
    private ExampleDbConnectionState state;

    @Override
    protected void onInit(Map<String, String> properties) throws DbException {
        registerOperationHandler(LdbcQuery1.class, LdbcQuery1Handler.class);
        registerOperationHandler(LdbcQuery2.class, LdbcQuery2Handler.class);
        state = new ExampleDbConnectionState(properties.get(‘‘url’’));
    }

    @Override
    protected void onCleanup() throws DbException {}

    @Override
    protected DbConnectionState getConnectionState() throws DbException {
        return state;
    }

    public static class LdbcQuery1Handler extends OperationHandler<LdbcQuery1> {
        @Override
```

```
        protected OperationResult executeOperation(LdbcQuery1 operation) throws DbException {
            Client client = ((ExampleDbConnectionState)dbConnectionState()).getClient();
            LdbcQuery1Result result = client.execute(operation)
            int resultCode = // typically used for debugging, e.g. to return error codes
            return operation.buildResult(resultCode, result);
        }
    }

    public static class LdbcQuery2ToHandler extends OperationHandler<LdbcQuery2> {
        @Override
        protected OperationResult executeOperation(LdbcQuery2 operation) throws DbException {
            Client client = ((ExampleDbConnectionState)dbConnectionState()).getClient();
            LdbcQuery2Result result = client.execute(operation)
            int resultCode = // typically used for debugging, e.g. returning error codes
            return operation.buildResult(resultCode, result);
        }
    }
}
```

General driver properties

Although a detailed explanation of the workload driver design is beyond the scope of this document, it is worth noting that it has a number of configuration parameters, these parameters are summarized in Table 3.2.1.

status	boolean	intermittently output workload status during execution
operationcount	integer	number of queries to generate
threadcount	integer	size of thread pool to use for query execution
resultfile	string	path to where workload results will be written
timeunit	enum	time unit to report result metrics in
timecompressionratio	double	adjust all query start times proportionally
gctdeltaduration	integer	min duration (ms) between dependent reads and writes
peeridentifiers	string[]	addresses of other driver processes (for distributed mode)
toleratedexecutiondelay	integer	max allowed time between scheduled and actual query start times
workload	string	class name of specific `Workload` implementation
database	string	class name of specific `Db` implementation

Table 3.4: General Driver Parameters

Note that default configuration files, with relevant parameter values set and required parameters commented, are provided in the driver distribution.

Workload specific driver properties

In addition to the general driver properties, the driver supports the definition of arbitrary properties, which are passed along to the pluggable driver components: `Db` and `Workload`. The first, `Db`, was described already in the previous section. The second is `Workload`, it is the class responsible for defining which operations, in which order and at what throughput, the driver will generate. From the point of view of the driver, the LDBC Social Network Benchmark is an implementation of the `Workload` class, in this case named `LdbcInteractiveWorkload`. For more details regarding the workload itself see 2.3.1.

The configuration parameters for the `LdbcInteractiveWorkload` workload are summarized in Table 3.2.1.

3.2.2 Running Workload

Finally, to run the workload execute the `main` method of the `Client` class in the driver. The general pattern for doing so is as follows:

parameter_dir	string	data generator parameters directory (read parameters)
data_dir	string	data generator data directory (dataset and write parameters)
LdbcQuery1_interleave	boolean	interval between successive query 1 executions
LdbcQuery2_interleave	boolean	interval between successive query 2 executions
LdbcQuery3_interleave	boolean	interval between successive query 3 executions
LdbcQuery4_interleave	boolean	interval between successive query 4 executions
LdbcQuery5_interleave	boolean	interval between successive query 5 executions
LdbcQuery6_interleave	boolean	interval between successive query 6 executions
LdbcQuery7_interleave	boolean	interval between successive query 7 executions
LdbcQuery8_interleave	boolean	interval between successive query 8 executions
LdbcQuery9_interleave	boolean	interval between successive query 9 executions
LdbcQuery10_interleave	boolean	interval between successive query 10 executions
LdbcQuery11_interleave	boolean	interval between successive query 11 executions
LdbcQuery12_interleave	boolean	interval between successive query 12 executions
LdbcQuery13_interleave	boolean	interval between successive query 13 executions
LdbcQuery14_interleave	integer	interval between successive query 14 executions
LdbcQuery1_enable	boolean	enable/disable read query 1
LdbcQuery2_enable	boolean	enable/disable read query 2
LdbcQuery3_enable	boolean	enable/disable read query 3
LdbcQuery4_enable	boolean	enable/disable read query 4
LdbcQuery5_enable	boolean	enable/disable read query 5
LdbcQuery6_enable	boolean	enable/disable read query 6
LdbcQuery7_enable	boolean	enable/disable read query 7
LdbcQuery8_enable	boolean	enable/disable read query 8
LdbcQuery9_enable	boolean	enable/disable read query 9
LdbcQuery10_enable	boolean	enable/disable read query 10
LdbcQuery11_enable	boolean	enable/disable read query 11
LdbcQuery12_enable	boolean	enable/disable read query 12
LdbcQuery13_enable	boolean	enable/disable read query 13
LdbcQuery14_enable	boolean	enable/disable read query 14
LdbcUpdate1AddPerson_enable	boolean	enable/disable update query 1
LdbcUpdate2AddPostLike_enable	boolean	enable/disable update query 2
LdbcUpdate3AddCommentLike_enable	boolean	enable/disable update query 3
LdbcUpdate4AddForum_enable	boolean	enable/disable update query 4
LdbcUpdate5AddForumMembership_enable	boolean	enable/disable update query 5
LdbcUpdate6AddPost_enablet	boolean	enable/disable update query 6
LdbcUpdate7AddComment_enable	boolean	enable/disable update query 7
LdbcUpdate8AddFriendship_enable	boolean	enable/disable update query 8

Table 3.5: LDBC Social Network Benchmark Parameters

```
usage: java -cp core-VERSION.jar com.ldbc.driver.Client [-db <classname>] [-del <duration>] [-gctd <duration>]
       [-oc <count>] [-P <file1:file2>] [-p <key=value>] [-pids <peerId1:peerId2>] [-rf <path>] [-s] [-tc
       <count>] [-tcr <ratio>] [-tu <unit>] [-w <classname>]
   -db,--database <classname>                    classname of the DB to use (e.g.
                                                 com.ldbc.driver.workloads.simple.db.BasicDb)
   -del,--toleratedexecutiondelay <duration>     duration (ms) an operation handler may miss its scheduled
                                                 start time by
   -gctd,--gctdeltaduration <duration>           safe duration (ms) between dependent operations
   -oc,--operationcount <count>                  number of operations to execute (default: 0)
   -P <file1:file2>                              load properties from file(s) - files will be loaded in the
```

```
                                          order provided
                                          first files are highest priority; later values will not
                                          override earlier values
   -p <key=value>                         properties to be passed to DB and Workload - these will
                                          override properties loaded from files
   -pids,--peeridentifiers <peerId1:peerId2>  identifiers/addresses of other driver workers (for
                                          distributed mode)
   -rf,--resultfile <path>                where benchmark results JSON file will be written (null =
                                          file will not be created)
   -s,--status                            show status during run
   -tc,--threadcount <count>              number of worker threads to execute with (default: 2)
   -tcr,--timecompressionratio <ratio>    change duration between operations of workload
   -tu,--timeunit <unit>                  time unit to use when gathering metrics.
                                          default:MILLISECONDS, valid:[NANOSECONDS, MICROSECONDS,
                                          MILLISECONDS, SECONDS, MINUTES]
   -w,--workload <classname>              classname of the Workload to use (e.g.
                                          com.ldbc.driver.workloads.simple.SimpleWorkload)
```

For the Interactive workload of LDBC-SNB it is necessary to provide the driver with four configuration files, containing: general driver properties, workload specific driver properties (e.g. database connection string), a dataset specific value for one of the general driver properties (specifically, `gctdeltaduration`), and vendor specific database properties. The commandline to execute the workload would look something like:

```
java -cp ldbc_driver/target/core-0.2-SNAPSHOT.jar com.ldbc.driver.Client
-db com.vendor.VendorDb
-P vendor/vendor.properties,
-P data_generator/outputDir/updateStream_0.properties,
-P ldbc_driver/workloads/ldbc/socnet/interactive/ldbc_socnet_interactive.properties
-P ldbc_driver/src/main/resources/ldbc_driver_default.properties
```

3.3 Gathering the results

Gathering the results for a benchmark run is a simple matter of referring to the results JSON file emitted by the driver. The format of that file is as follows:

```
{
  "unit": "MILLISECONDS",
  "start_time": 1400750662691,
  "finish_time": 1400750667691,
  "total_duration": 5000,
  "total_count": 50,
  "all_metrics": [
    {
      "name": "Query1",
      "count": 50,
      "unit": "MILLISECONDS",
      "run_time": {
        "name": "Runtime",
        "unit": "MILLISECONDS",
        "count": 50,
        "mean": 100,
        "min": 2,
        "max": 450,
        "50th_percentile": 98,
        "90th_percentile": 129,
        "95th_percentile": 432,
        "99th_percentile": 444
      },
      "start_time_delay": {
        "name": "Start Time Delay",
```

```
        "unit": "MILLISECONDS",
        "count": 7,
        "mean": 3.5714285714285716,
        "min": 0,
        "max": 25,
        "50th_percentile": 0,
        "90th_percentile": 0,
        "95th_percentile": 25,
        "99th_percentile": 25
      },
      "result_code": {
        "name": "Result Code",
        "unit": "Result Code",
        "count": 50,
        "all_values": {
          "0": 42,
          "1": 8
        }
      }
    }
  ]
}
```

REFERENCES

[1] Alex Averbuch. https://github.com/ldbc/ldbc_driver, June 2014.

[2] Jure Leskovec, Lars Backstrom, Ravi Kumar, and Andrew Tomkins. Microscopic evolution of social networks. In *KDD*, pages 462–470, 2008.

[3] M. McPherson, L. Smith-Lovin, and J. M. Cook. Birds of a feather: Homophily in social networks. *Annual review of sociology*, pages 415–444, 2001.

[4] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The anatomy of the facebook social graph. *CoRR*, abs/1111.4503, 2011.

A Interactive Query Set Implementations

A.1 Virtuoso SPARQL 1.1

A.1.1 Query 1

```
sparql select ?fr ?last min(?dist) as ?mindist  ?bday ?since ?gen ?browser ?locationIP
   ((select group_concat (?email, ", ")
       where {
           ?frr snvoc:email ?email .
       filter (?frr = ?fr) .
       }
       group by ?frr)) as ?email
   ((select group_concat (?lng, ", ")
       where {
           ?frr snvoc:speaks ?lng .
       filter (?frr = ?fr) .
       }
       group by ?frr)) as ?lng
   ?based
   ((select group_concat ( bif:concat (?o_name, " ", ?year, " ", ?o_country), ", ")
       where {
       ?frr snvoc:studyAt ?w .
       ?w snvoc:classYear ?year .
           ?w snvoc:hasOrganisation ?org .
           ?org snvoc:isLocatedIn ?o_countryURI .
       ?o_countryURI foaf:name ?o_country .
           ?org foaf:name ?o_name .
       filter (?frr = ?fr) .
       }
       group by ?frr)) as ?studyAt
   ((select group_concat ( bif:concat (?o_name, " ", ?year, " ", ?o_country), ", ")
       where {
       ?frr snvoc:workAt ?w .
       ?w snvoc:workFrom ?year .
           ?w snvoc:hasOrganisation ?org .
           ?org snvoc:isLocatedIn ?o_countryURI .
       ?o_countryURI foaf:name ?o_country .
           ?org foaf:name ?o_name .
       filter (?frr = ?fr) .
       }
       group by ?frr)) as ?workAt
   {
       ?fr a snvoc:Person .
       ?fr snvoc:firstName "%Name%" .
   ?fr snvoc:lastName ?last .
       ?fr snvoc:birthday ?bday .
       ?fr snvoc:isLocatedIn ?basedURI .
   ?basedURI foaf:name ?based .
       ?fr snvoc:creationDate ?since .
       ?fr snvoc:gender ?gen .
```

```
    ?fr snvoc:locationIP ?locationIP .
    ?fr snvoc:browserUsed ?browser .

        {
          { select distinct ?fr (1 as ?dist)
            where {
              sn:pers%Person% snvoc:knows ?fr.
            }
          }
      union
          { select distinct ?fr (2 as ?dist)
            where {
              sn:pers%Person% snvoc:knows ?fr2.
                              ?fr2 snvoc:knows ?fr.
                              filter (?fr != sn:pers%Person%).
            }
          }
      union
          { select distinct ?fr (3 as ?dist)
            where {
              sn:pers%Person% snvoc:knows ?fr2.
                              ?fr2 snvoc:knows ?fr3.
                              ?fr3 snvoc:knows ?fr.
                              filter (?fr != sn:pers%Person%).
            }
          } .
        }
    }
    group by ?fr ?last ?bday ?since ?gen ?browser ?locationIP ?based
    order by ?mindist ?last ?fr
    limit 20
```

A.1.2 Query 2

```
 sparql select ?fr ?first ?last ?post ?content ?date
from <sib>
where {
  sn:pers%Person% snvoc:knows ?fr.
  ?fr snvoc:firstName ?first. ?fr snvoc:lastName ?last .
  ?post snvoc:hasCreator ?fr.
  { {?post snvoc:content ?content } union { ?post snvoc:imageFile ?content }} .
  ?post snvoc:creationDate ?date.
  filter (?date <= "%Date0%"^^xsd:date).
}
order by desc (?date) ?post
limit 20
```

A.1.3 Query 3

```
 sparql select ?fr ?first ?last ?ct1 ?ct2 (?ct1 + ?ct2) as ?sum
from <sib>
where {
```

```
    {select distinct ?fr ?first ?last
        (((select count (*)
        where {
            ?post snvoc:hasCreator ?fr .
            ?post snvoc:creationDate ?date .
            filter (?date >= "%Date0%"^^xsd:date &&
                    ?date < bif:dateadd ("day", %Duration%, "%Date0%"^^xsd:date)) .
            ?post snvoc:isLocatedIn dbpedia:%Country1%
        }))
        as ?ct1)
        ((select count (*)
        where {
            ?post2 snvoc:hasCreator ?fr .
            ?post2 snvoc:creationDate ?date2 .
            filter (?date2 >= "%Date0%"^^xsd:date &&
                    ?date2 < bif:dateadd ("day", %Duration%, "%Date0%"^^xsd:date)) .
            ?post2 snvoc:isLocatedIn dbpedia:%Country2%
        })
        as ?ct2)
    where {
        {sn:pers%Person% snvoc:knows ?fr.} union { sn:pers%Person% snvoc:knows ?fr2.
                                                   ?fr2 snvoc:knows ?fr.
                                                   filter (?fr != sn:pers%Person%)
                                                 }.
        ?fr snvoc:firstName ?first . ?fr snvoc:lastName ?last .
        ?fr snvoc:isLocatedIn ?city .
    filter(!exists {?city snvoc:isPartOf dbpedia:%Country1%}).
    filter(!exists {?city snvoc:isPartOf dbpedia:%Country2%}).
    }
    }.
    filter (?ct1 > 0 && ?ct2 > 0) .
}
order by desc(6) ?fr
limit 20
```

A.1.4 Query 4

```
sparql select ?tagname count (*)
from <sib>
where {
    ?post snvoc:hasCreator ?fr .
    ?post snvoc:hasTag ?tag .
    ?tag foaf:name ?tagname .
    ?post snvoc:creationDate ?date .
    sn:pers%Person% snvoc:knows ?fr .
    filter (?date >= "%Date0%"^^xsd:date &&
            ?date <= bif:dateadd ("day", %Duration%, "%Date0%"^^xsd:date) ) .
    filter (!exists {
        sn:pers%Person% snvoc:knows ?fr2 .
        ?post2 snvoc:hasCreator ?fr2 .
        ?post2 snvoc:hasTag ?tag .
```

```
        ?post2 snvoc:creationDate ?date2 .
        filter (?date2 < "%Date0%"^^xsd:date)})
    }
group by ?tagname
order by desc(2) ?tagname
limit 10
```

A.1.5 Query 5

```
sparql select ?title count (*)
from <sib>
where {
    {select distinct ?fr
     from <sib>
     where {
       {sn:pers%Person% snvoc:knows ?fr.} union { sn:pers%Person% snvoc:knows ?fr2.
                                                  ?fr2 snvoc:knows ?fr.
                                                  filter (?fr != sn:pers%Person%) }
     }
    } .
    ?group snvoc:hasMember ?mem .
    ?mem snvoc:hasPerson ?fr .
    ?mem snvoc:joinDate ?date .
    filter (?date >= "%Date0%"^^xsd:date) .
    ?post snvoc:hasCreator ?fr .
    ?group snvoc:containerOf ?post .
    ?group snvoc:title ?title.
}
group by ?title
order by desc(2) ?title
limit 20
```

A.1.6 Query 6

```
sparql select ?tagname count (*)
from <sib>
where {
    {  select distinct ?fr
       from <sib>
       where {
           {sn:pers%Person% snvoc:knows ?fr.} union { sn:pers%Person% snvoc:knows ?fr2.
                                                      ?fr2 snvoc:knows ?fr.
                                                      filter (?fr != sn:pers%Person%) }
       }
    } .
    ?post snvoc:hasCreator ?fr .
    ?post snvoc:hasTag ?tag1 .
    ?tag1 foaf:name ?tagname1 .
    filter (?tagname1 != '%Tag%') .
    ?post snvoc:hasTag ?tag .
    ?tag foaf:name ?tagname .
}
```

```
group by ?tagname
order by desc(2) ?tagname
limit 10
```

A.1.7 Query 7

```
 sparql select ?liker ?first ?last ?ldt
       (if ((exists {  sn:pers%Person% snvoc:knows ?liker}), 0, 1) as ?is_new)
       ?post ?content (bif:datediff ("minute", ?dt, ?ldt) as ?lag)
from <sib>
where {
  ?post snvoc:hasCreator sn:pers%Person% .
  {{ ?post snvoc:content ?content } union {?post snvoc:imageFile ?content}} .
  ?lk snvoc:hasPost ?post .
  ?liker snvoc:likes ?lk . ?liker  snvoc:firstName ?first . ?liker snvoc:lastName ?last .
  ?post snvoc:creationDate ?dt . ?lk snvoc:creationDate ?ldt .
}
order by desc (?ldt) ?liker
limit 20
```

A.1.8 Query 8

```
sparql select ?from ?first ?last ?dt ?rep ?content
where {
  { select ?rep ?dt
    where {
        ?post snvoc:hasCreator sn:pers%Person% .
        ?rep snvoc:replyOf ?post . ?rep snvoc:creationDate ?dt .
    }
    order by desc (?dt)
    limit 20
  } .
  ?rep snvoc:hasCreator ?from .
  ?from snvoc:firstName ?first . ?from snvoc:lastName ?last .
  ?rep snvoc:content ?content.
}
order by desc(?dt) ?rep
```

A.1.9 Query 9

```
sparql select ?fr ?first ?last ?post ?content ?date
from <sib>
where {
  {select distinct ?fr
   from <sib>
   where {
       {sn:pers%Person% snvoc:knows ?fr.} union { sn:pers%Person% snvoc:knows ?fr2.
                                                  ?fr2 snvoc:knows ?fr.
                                                  filter (?fr != sn:pers%Person%) }
   }
  }
  ?fr snvoc:firstName ?first . ?fr snvoc:lastName ?last .
```

```
  ?post snvoc:hasCreator ?fr.
  ?post snvoc:creationDate ?date.
  filter (?date < "%Date0%"^^xsd:date).
  {{?post snvoc:content ?content} union {?post snvoc:imageFile ?content}} .
}
order by desc (?date) ?post
limit 20
```

A.1.10 Query 10

```
 sparql select ?first ?last
    ((( select  count (distinct ?post)
        where {
            ?post snvoc:hasCreator ?fof .
            ?post snvoc:hasTag ?tag .
            sn:pers%Person% snvoc:hasInterest ?tag
        }
    ))
    -
    ((  select  count (distinct ?post)
        where {
            ?post snvoc:hasCreator ?fof .
            ?post snvoc:hasTag ?tag .
            filter (!exists {sn:pers%Person% snvoc:hasInterest ?tag})
        }
    )) as ?score)
    ?fof  ?gender ?locationname
from <sib>
where {
   {select distinct ?fof
    where {
        sn:pers%Person% snvoc:knows ?fr .
        ?fr snvoc:knows ?fof .
    filter (?fof != sn:pers%Person%)
        minus { sn:pers%Person% snvoc:knows ?fof } .
    }
   } .
   ?fof snvoc:firstName ?first .
   ?fof snvoc:lastName ?last .
   ?fof snvoc:gender ?gender .
   ?fof snvoc:birthday ?bday .
   ?fof snvoc:isLocatedIn ?based .
   ?based foaf:name ?locationname .
   filter (1 = if (bif:month (?bday) = %HS0%, if (bif:dayofmonth (?bday) > 21, 1, 0),
               if (bif:month (?bday) = %HS1%, if (bif:dayofmonth(?bday) < 22, 1, 0), 0)))
}
order by desc(3) ?fof
limit 10
```

A.1.11 Query 11

```
 sparql select ?first ?last ?startdate ?orgname ?fr
```

```
where {
    ?w snvoc:hasOrganisation ?org .
    ?org foaf:name ?orgname .
    ?org snvoc:isLocatedIn ?country.
    ?country foaf:name '%Country%' .
    ?fr snvoc:workAt ?w .
    ?w snvoc:workFrom ?startdate .
    filter (?startdate < %Date0%) .
    {  select distinct ?fr
       from <sib>
       where {
           {sn:pers%Person% snvoc:knows ?fr.} union { sn:pers%Person% snvoc:knows ?fr2.
                                                      ?fr2 snvoc:knows ?fr.
                                                      filter (?fr != sn:pers%Person%) }
       }
    } .
    ?fr snvoc:firstName ?first .
    ?fr snvoc:lastName ?last .
}
order by ?startdate ?fr ?orgname
limit 10
```

A.1.12 Query 12

```
sparql select ?exp ?first ?last sql:group_concat_distinct(?tagname) count (*) #Q12
where {
    sn:pers%Person% snvoc:knows ?exp .
    ?exp snvoc:firstName ?first . ?exp snvoc:lastName ?last .
    ?reply snvoc:hasCreator ?exp .
    ?reply snvoc:replyOf  ?org_post .
    filter (!exists {?org_post snvoc:replyOf ?xx}) .
    ?org_post snvoc:hasTag ?tag .
    ?tag foaf:name ?tagname .
    ?tag a ?type.
    ?type rdfs:subClassOf* ?type1 .
    ?type1 rdfs:label %TagType% .
}
group by ?exp ?first ?last
order by desc(5) ?exp
limit 20
```

A.1.13 Query 13

```
sparql select count(*)
where
  {
    {
      select ?s ?o
      where
        {
          ?s snvoc:knows ?o.
        }
```

```
    }
    option ( transitive,
             t_distinct,
             t_in(?s),
             t_out(?o),
             t_shortest_only,
             t_direction 3,
             t_step ('path_id') as ?path_no) .
    filter ( ?s = sn:pers%Person1% ).
    filter ( ?o = sn:pers%Person2% ).
    filter (?path_no = 0).
  }
```

A.1.14 Query 14

```
create procedure path_str_sparql (in path any)
{
  declare str any;
  declare inx int;
  str := '';
  foreach (any  st  in path) do
    str := str || sprintf (' %d->%d (%d) ',
                           cast (substring(st[0], 48, 20) as int),
                           coalesce(cast (substring(st[1], 48, 20) as int), 0),
                           coalesce (st[2], 0));
  return str;
}

create procedure c_weight_sparql (in p1 varchar, in p2 varchar)
{
  vectored;
  if (p1 is null or p2 is null)
     return 0;
  return 0.5 +
       ( sparql select count(*) from <sib> where {?post1 snvoc:hasCreator ?:p1.
                                                  ?post1 snvoc:replyOf ?post2.
                                                  ?post2 snvoc:hasCreator ?:p2.
                                                  ?post2 a snvoc:Post} ) +
       ( sparql select count(*) from <sib> where {?post1 snvoc:hasCreator ?:p2.
                                                  ?post1 snvoc:replyOf ?post2.
                                                  ?post2 snvoc:hasCreator ?:p1.
                                                  ?post2 a snvoc:Post} ) +
       ( sparql select 0.5 * count(*) from <sib> where {?post1 snvoc:hasCreator ?:p1.
                                                        ?post1 snvoc:replyOf ?post2.
                                                        ?post2 snvoc:hasCreator ?:p2.
                                                        ?post2 a snvoc:Comment} ) +
       ( sparql select 0.5 * count(*) from <sib> where {?post1 snvoc:hasCreator ?:p2.
                                                        ?post1 snvoc:replyOf ?post2.
                                                        ?post2 snvoc:hasCreator ?:p1.
                                                        ?post2 a snvoc:Comment} );
}
```

```
select sql:path_str_sparql(?path), ?sc
where
{
  select ?path_no, sql:vector_agg (bif:vector (?via1, ?via2, ?cweight))
                                    as ?path, sum (?cweight)
                                    as ?sc
  where
  {
    select ?via1 ?via2 ?path_no ?step_no sql:c_weight_sparql(?via1, ?via2) as ?cweight
    where
    {
      {
        select ?s bif:idn(?s) as ?via2 ?o
        where
        {
          ?s snvoc:knows ?o1.
      ?o1 snvoc:hasPerson ?o .
        }
      }
      option ( transitive,
             t_distinct,
             t_in(?s),
             t_out(?o),
         t_shortest_only,
         t_direction 3,
         t_step (?s) as ?via1,
         t_step ('path_id') as ?path_no,
             t_step ('step_no') as ?step_no ) .
      filter ( ?s = %Person1% ).
      filter ( ?o = %Person2% ).
    }
  }
  group by ?path_no
}
order by desc(?sc)
limit 10
```

A.2 Virtuoso SQL

A.2.1 Query 1

```
select top 20 id, p_lastname, min (dist) as dist,
       p_birthday, p_creationdate, p_gender, p_browserused,
       bit_shift(bit_and(p_locationip, 4278190080), -24) || '.' ||
       bit_shift(bit_and(p_locationip, 16711680), -16) || '.' ||
       bit_shift(bit_and(p_locationip, 65280), -8) || '.' ||
       bit_and(p_locationip, 255) as ip,
       (select group_concat (pe_email, ', ')
            from person_email
            where pe_personid = id
```

```
          group by pe_personid) as emails,
       (select group_concat (plang_language, ', ')
          from person_language
          where plang_personid = id
          group by plang_personid) as languages,
       p1.pl_name,
       (select group_concat (o2.o_name || ' ' || pu_classyear || ' ' || p2.pl_name, ', ')
          from person_university, organisation o2, place p2
          where pu_personid = id and
                pu_organisationid = o2.o_organisationid and
                o2.o_placeid = p2.pl_placeid
          group by pu_personid) as university,
       (select group_concat (o3.o_name || ' ' || pc_workfrom || ' ' || p3.pl_name, ', ')
          from person_company, organisation o3, place p3
          where pc_personid = id and
                pc_organisationid = o3.o_organisationid and
                o3.o_placeid = p3.pl_placeid
          group by pc_personid) as company
from
    (
    select k_person2id as id, 1 as dist from knows, person
                                       where k_person1id = @Person@ and
                                             p_personid = k_person2id and
                                             p_firstname = '@Name@'
    union all
    select b.k_person2id as id, 2 as dist from knows a, knows b, person
    where
      a.k_person1id = @Person@ and
      b.k_person1id = a.k_person2id and
      p_personid = b.k_person2id and
      p_firstname = '@Name@'
    union all
    select c.k_person2id as id, 3 as dist from knows a, knows b, knows c, person
    where
      a.k_person1id = @Person@ and
      b.k_person1id = a.k_person2id and
      b.k_person2id = c.k_person1id and
      p_personid = c.k_person2id and
      p_firstname = '@Name@'
    ) tmp, person, place p1
  where
    p_personid = id and
    p_placeid = p1.pl_placeid
  group by id, p_lastname
  order by dist, p_lastname, id
```

A.2.2 Query 2

```
select top 20 p_personid as personid, p_firstname as firstname, p_lastname as lastname,
       ps_postid as id, ps_content || ps_imagefile as content, ps_creationdate as creationdate
from person, post, knows
```

```
where
    p_personid = ps_creatorid and
    ps_creationdate <= stringdate('@Date0@') and
    k_person1id = @Person@ and
    k_person2id = p_personid
order by creationdate desc, id
```

A.2.3 Query 3

```
select top 20 p_personid, p_firstname, p_lastname, ct1, ct2, total
from
 ( select k_person2id
   from knows
   where
   k_person1id = @Person@
   union
   select k2.k_person2id
   from knows k1, knows k2
   where k1.k_person1id = @Person@ and
         k1.k_person2id = k2.k_person1id and
         k2.k_person2id <> @Person@
 ) f,  person, place p1, place p2,
 (
  select chn.ps_c_creatorid, ct1, ct2, ct1 + ct2 as total
  from
   (
      select ps_creatorid as ps_c_creatorid, count(*) as ct1
      from post, place
      where  ps_locationid = pl_placeid and
             pl_name = '@Country1@' and
             ps_creationdate between stringdate('@Date0@') and
             dateadd ('day', @Duration, stringdate('@Date0@'))
      group by ps_c_creatorid
   ) chn,
   (
      select ps_creatorid as ps_c_creatorid, count(*) as ct2
      from post, place
      where ps_locationid = pl_placeid and
            pl_name = '@Country2@' and
            ps_creationdate between stringdate('@Date0@') and
            dateadd ('day', 366, stringdate('@Date0@'))
      group by ps_c_creatorid
   ) ind
  where CHN.ps_c_creatorid = IND.ps_c_creatorid
 ) cpc
where f.k_person2id = p_personid and
      p_placeid = p1.pl_placeid and
      p1.pl_containerplaceid = p2.pl_placeid and
      p2.pl_name <> '@Country1@' and
      p2.pl_name <> '@Country2@' and
      f.k_person2id = cpc.ps_c_creatorid
```

```
order by 6 desc, 1
```

A.2.4 Query 4

```
select top 10 t_name, count(*)
from tag, post, post_tag, knows
where
    ps_postid = pst_postid and
    pst_tagid = t_tagid and
    ps_creatorid = k_person2id and
    k_person1id = @Person@ and
    ps_creationdate between stringdate('@Date0@') and
    dateadd ('day', @Duration@, stringdate('@Date0@')) and
    not exists (
        select * from post, post_tag, knows
        where
        k_person1id = @Person@ and
        k_person2id = ps_creatorid and
        pst_postid = ps_postid and
        pst_tagid = t_tagid and
        ps_creationdate < '@Date0@'
    )
group by t_name
order by 2 desc, t_name
```

A.2.5 Query 5

```
select top 20 f_title, count(*)
from forum, post, forum_person,
 ( select k_person2id
   from knows
   where
   k_person1id = @Person@
   union
   select k2.k_person2id
   from knows k1, knows k2
   where
   k1.k_person1id = @Person@ and
   k1.k_person2id = k2.k_person1id and
   k2.k_person2id <> @Person@
 ) f
where f_forumid = ps_forumid and
      f_forumid = fp_forumid and
      fp_personid = f.k_person2id and
      ps_creatorid = f.k_person2id and
      fp_creationdate >= stringdate('@Date0@')
group by f_title
order by 2 desc, f_title
```

A.2.6 Query 6

```
select top 10 t_name, count(*)
```

```
from tag, post_tag, post,
 ( select k_person2id
   from knows
   where
   k_person1id = @Person@
   union
   select k2.k_person2id
   from knows k1, knows k2
   where k1.k_person1id = @Person@ and
         k1.k_person2id = k2.k_person1id and
         k2.k_person2id <> @Person@
 ) f
where
ps_creatorid = f.k_person2id and
ps_postid = pst_postid and
pst_tagid = t_tagid and
t_name <> '@Tag@' and
exists (select *
        from tag, post_tag
        where pst_postid = ps_postid and
              pst_tagid = t_tagid and
              t_name = '@Tag@')
group by t_name
order by 2 desc, t_name
```

A.2.7 Query 7

```
select top 20 p_personid , p_firstname, p_lastname, l_creationdate,
              (case when k_person2id is null then 1 else 0 end) as is_new,
              ps_postid, content, lag
from
(select p_personid, p_firstname, p_lastname, l_creationdate,
        ps_postid, ps_content || ps_imagefile as content,
    datediff('minute', ps_creationdate, l_creationdate) as lag
from likes, post, person
where
    p_personid = l_personid and
    ps_postid = l_postid and
    ps_creatorid = @Person@
) p
left join
(select * from knows where k_person1id = @Person@) k
on k.k_person2id = p.p_personid
order by l_creationdate desc, 1
```

A.2.8 Query 8

```
select top 20 p1.ps_creatorid,
              p_firstname,
              p_lastname,
              p1.ps_creationdate,
              p1.ps_postid,
```

```
              p1.ps_content
  from post p1, post p2, person
  where
      p1.ps_replyof = p2.ps_postid and
      p2.ps_creatorid = @Person@ and
      p_personid = p1.ps_creatorid
order by p1.ps_creationdate desc, 5
```

A.2.9 Query 9

```
select top 20 p_personid, p_firstname, p_lastname,
       ps_postid, ps_content || ps_imagefile, ps_creationdate
from person, post,
  ( select k_person2id
    from knows
    where
    k_person1id = @Person@
    union
    select k2.k_person2id
    from knows k1, knows k2
    where k1.k_person1id = @Person@ and
          k1.k_person2id = k2.k_person1id and
          k2.k_person2id <> @Person@
  ) f
where
  p_personid = ps_creatorid and p_personid = f.k_person2id and
  ps_creationdate < stringdate('@Date0@')
order by ps_creationdate desc, 4
```

A.2.10 Query 10

```
select top 10 p_firstname, p_lastname,
       ( select count(distinct ps_postid)
         from post, post_tag pt1
         where ps_creatorid = p_personid and
               ps_postid = pst_postid and
     exists (select * from person_tag
                      where pt_personid = @Person@ and
                            pt_tagid = pt1.pst_tagid)
       ) -
       ( select count(distinct ps_postid)
         from post, post_tag pt1
         where ps_creatorid = p_personid and
               ps_postid = pst_postid and
     not exists (select * from person_tag
                          where pt_personid = @Person@ and
                                pt_tagid = pt1.pst_tagid)
       ) as score,
       p_personid, p_gender, pl_name
from person, place,
 ( select distinct k2.k_person2id
   from knows k1, knows k2
```

```
   where k1.k_person1id = @Person@ and
         k1.k_person2id = k2.k_person1id and
         k2.k_person2id <> @Person@ and
   not exists (select * from knows
                        where k_person1id = @Person@ and
                              k_person2id = k2.k_person2id)
 ) f
where
p_placeid = pl_placeid and
p_personid = f.k_person2id and
case month(p_birthday)
    when @HS0@ then (case when dayofmonth(p_birthday) > 21 then 1 else 0 end)
    when @HS1@ then (case when dayofmonth(p_birthday) < 22 then 1 else 0 end)
    else 0
end
order by 3 desc, 4
```

A.2.11 Query 11

```
select top 10 p_firstname, p_lastname, pc_workfrom, o_name, p_personid
from person, person_company, organisation, place,
 ( select k_person2id
   from knows
   where
   k_person1id = @Person@
   union
   select k2.k_person2id
   from knows k1, knows k2
   where k1.k_person1id = @Person@ and
         k1.k_person2id = k2.k_person1id and
         k2.k_person2id <> @Person@
 ) f
where
    p_personid = f.k_person2id and
    p_personid = pc_personid and
    pc_organisationid = o_organisationid and
    pc_workfrom < @Date0@ and
    o_placeid = pl_placeid and
    pl_name = '@Country@'
order by pc_workfrom, 5, o_name
```

A.2.12 Query 12

```
select top 20   p_personid,
                p_firstname,
                p_lastname,
group_concat_distinct(t_name, ', '), count(*)
from person, post p1, knows, post p2, post_tag, tag, tag_tagclass
where
  k_person1id = @Person@ and
  k_person2id = p_personid and
  p_personid = p1.ps_creatorid and
```

```
  p1.ps_replyof = p2.ps_postid and
  p2.ps_replyof is null and
  p2.ps_postid = pst_postid and
  pst_tagid = t_tagid and
  t_tagid = ttc_tagid and
  (ttc_tagclassid in (
           select s_subtagclassid from
             (select transitive t_in (1)
                                t_out (2)
                                t_distinct
                                s_subtagclassid,
                                s_supertagclassid
             from subclass) k, tagclass
         where tc_tagclassid = k.s_supertagclassid and tc_name = '@TagType@'
         )
   or
   ttc_tagclassid = (select tc_tagclassid from tagclass where tc_name = '@TagType@')
   )
group by 1, p_firstname, p_lastname
order by 5 desc, 1
```

A.2.13 Query 13

```
select count(*)
from
  (select transitive t_in (1)
                     t_out (2)
                     t_distinct
                     t_shortest_only
                     t_direction 3
   k_person1id as p1, k_person2id as p2, t_step ('path_id') as path_no from knows) kt
where
  p1 = @Person1@ and
  p2 = @Person2@ and
  path_no = 0
```

A.2.14 Query 14

```
create procedure path_str (in path any)
{
  declare str any;
  declare inx int;
  str := '';
  foreach (any  st  in path) do
    str := str || sprintf (' %d->%d (%d) ', st[0], coalesce (st[1], 0), coalesce (st[2], 0));
  return str;
}
create procedure c_weight (in p1 bigint, in p2 bigint)
{
  vectored;
  if (p1 is null or p2 is null)
     return 0;
```

```
  return 0.5 +
       (select count (*)
            from post ps1, post ps2
            where ps1.ps_creatorid = p1 and
                  ps1.ps_replyof = ps2.ps_postid and
                  ps2.ps_creatorid = p2 and
                  ps2.ps_replyof is null) +
       (select count (*) from post ps1, post ps2
            where ps1.ps_creatorid = p2 and
                  ps1.ps_replyof = ps2.ps_postid and
                  ps2.ps_creatorid = p1 and
                  ps2.ps_replyof is null) +
       (select 0.5 * count (*)
            from post c1, post c2
            where c1.ps_creatorid = p1 and
                  c1.ps_replyof = c2.ps_postid and
                  c2.ps_creatorid = p2 and
                  c2.ps_replyof is not null) +
       (select 0.5 * count (*)
            from post c1, post c2
            where c1.ps_creatorid = p2 and
                  c1.ps_replyof = c2.ps_postid and
                  c2.ps_creatorid = p1 and
                  c2.ps_replyof is not null);
}
select top 10 path_str (path), sc
from
  (select path_no, vector_agg (vector (via1, via2, cweight)) as path, sum (cweight) as sc
   from
       (select path_no, step_no, via1, via2, c_weight (via1, via2) as cweight
        from
          (select transitive t_in (1)
                             t_out (2)
                             t_distinct
                             t_shortest_only
                             t_direction 3
                  k_person1id as p1,
                  k_person2id as p2,
                  t_step (1) as via1, idn (k_person1id) as via2,
                  t_step ('path_id') as path_no, t_step ('step_no') as step_no from knows) kt
        where p1 = @Person1@ and p2 = @Person2@) w
   group by path_no) paths
order by sc desc
```

A.3 Neo Cypher

A.3.1 Query 1

```
MATCH (:Person {id:{person_id}})-[path:KNOWS*]-(friend:Person)
WHERE friend.firstName = {friend_first_name}
WITH friend, min(length(path)) AS distance
```

```
ORDER BY distance ASC, friend.lastName ASC
LIMIT {limit}
OPTIONAL MATCH (friend)-[:IS_LOCATED_IN]->(friendCity:City)
OPTIONAL MATCH (friend)-[studyAt:STUDY_AT]->
               (uni:University)-[:IS_LOCATED_IN]->
               (uniCity:City)
WITH friend, collect(uni.name +
                     ',' +
                     uniCity.name +
                     ',' +
                     studyAt.classYear) AS unis,
                                        friendCity,
                                        distance
OPTIONAL MATCH (friend)-[worksAt:WORKS_AT]->
               (company:Company)-[:IS_LOCATED_IN]->
               (companyCountry:Place:Country)
WITH friend,
     collect(company.name +
             ',' +
             companyCountry.name +
             ',' +
             worksAt.workFrom)
AS companies,
   unis,
   friendCity,
   distance

RETURN friend.id AS id,
       friend.lastName AS lastName,
       distance,
       friend.birthday AS birthday,
       friend.creationDate AS creationDate,
       friend.gender AS gender,
       friend.browserUsed AS browser,
       friend.locationIP AS locationIp,
       friend.email AS emails,
       friend.languages AS languages,
       friendCity.name AS cityName,
       unis,
       companies
ORDER BY distance ASC, friend.lastName ASC
LIMIT {limit}
```

A.3.2 Query 2

```
MATCH (:Person {id:{person_id}})-[:KNOWS]-(friend:Person)<-[:HAS_CREATOR]-(post:Post)
WHERE post.creationDate <= {max_date}
RETURN friend.id AS personId,
       friend.firstName AS personFirstName,
       friend.lastName AS personLastName,
       post.id AS postId,
```

```
       post.content AS postContent,
       post.creationDate AS postDate
ORDER BY postDate DESC
LIMIT {limit}
```

A.3.3 Query 3

```
MATCH (person:Person {id:{person_id}})-[:KNOWS*1..2]-
                        (friend:Person)<-[:HAS_CREATOR]-
                        (postX:Post)-[:IS_LOCATED_IN]->
                        (countryX:Country)
WHERE countryX.name={country_x} AND
      postX.creationDate>={min_date} AND
      postX.creationDate<={max_date}
WITH friend, count(DISTINCT postX) AS xCount
MATCH (friend)<-[:HAS_CREATOR]-(postY:Post)-[:IS_LOCATED_IN]->
      (countryY:Country {name:{country_y}})
WHERE postY.creationDate>={min_date} AND postY.creationDate<={max_date}
WITH friend.firstName + ' ' + friend.lastName AS friendName ,
     xCount, count(DISTINCT postY) AS yCount
RETURN friendName, xCount, yCount, xCount + yCount AS xyCount
ORDER BY xyCount DESC
LIMIT {limit}
```

A.3.4 Query 4

```
MATCH (person:Person {id:{person_id}})-[:KNOWS]-
      (friend:Person)<-[:HAS_CREATOR]-
      (post:Post)-[HAS_TAG]->(tag:Tag)
WHERE post.creationDate >= {min_date} AND post.creationDate <= {max_date}
WITH DISTINCT tag, collect(tag) AS tags
RETURN tag.name AS tagName, length(tags) AS tagCount
ORDER BY tagCount DESC
LIMIT {limit}
```

A.3.5 Query 5

```
MATCH (person:Person {id:{person_id}})-[:KNOWS*1..2]-
      (friend:Person)<-[membership:HAS_MEMBER]-(forum:Forum)
WHERE membership.joinDate>{join_date}
MATCH (friend)<-[:HAS_CREATOR]-(post:Post)<-[:CONTAINER_OF]-(forum)
RETURN forum.title AS forum, count(post) AS postCount
ORDER BY postCount DESC
```

A.3.6 Query 6

```
MATCH (person:Person {id:{person_id}})-[:KNOWS*1..2]-
      (:Person)<-[:HAS_CREATOR]-
      (post:Post)-[:HAS_TAG]->(:Tag {name:{tag_name}})
WITH DISTINCT post
MATCH (post)-[:HAS_TAG]->(tag:Tag)
WHERE NOT(tag.name={tag_name})
```

```
RETURN tag.name AS tagName, count(tag) AS tagCount
ORDER BY tagCount DESC
LIMIT {limit}
```

A.3.7 Query 7

```
MATCH (start:Person {id:{person_id}})<-[:HAS_CREATOR]-
      (post:Post)<-[like:LIKES]-(person:Person)
RETURN person.id AS personId,
       person.firstName AS personFirstName,
       person.lastName AS personLastName,
       like.creationDate AS likeDate,
       NOT((person)-[:KNOWS]-(start)) AS isNew,
       post.id AS postId,
  post.content AS postContent, like.creationDate - post.creationDate AS latency
ORDER BY like.creationDate DESC, personId ASC
LIMIT {limit}
```

A.3.8 Query 8

```
MATCH (:Person {id:{person_id}})<-[:HAS_CREATOR]-(post:Post)
MATCH (post)<-[:REPLY_OF*]-(comment:Comment)-[:HAS_CREATOR]->(person:Person)
RETURN person.id AS personId,
       person.firstName AS personFirstName,
       person.lastName AS personLastName,
       comment.id AS commentId,
       comment.creationDate AS commentCreationDate,
       comment.content AS commentContent
ORDER BY commentCreationDate DESC, commentId ASC
LIMIT {limit}
```

A.3.9 Query 9

```
MATCH (:Person {id:{person_id}})-[:KNOWS*1..2]-(friend:Person)
MATCH (friend)<-[:HAS_CREATOR]-(activity) WHERE activity.creationDate < {latest_date}
RETURN DISTINCT activity.id AS activityId,
                activity.content AS activityContent,
                activity.creationDate AS activityCreationDate,
                friend.id AS personId,
                friend.firstName AS personFirstName,
                friend.lastName AS personLastName
ORDER BY activity.creationDate DESC, activity.id ASC
LIMIT {limit}
```

A.3.10 Query 10

```
MATCH (person:Person {id:{person_id}})
MATCH (person)-[:KNOWS*2..2]-(friend:Person)-[:IS_LOCATED_IN]->(city:City)
WHERE friend.birthday_month >= {horoscope_month_min} AND
      friend.birthday_month < {horoscope_month_max}
OPTIONAL MATCH (friend)<-[:HAS_CREATOR]-(post:Post)
WITH friend, city.name AS personCityName, count(post) AS allPostCount, person
```

```
OPTIONAL MATCH (friend)<-[:HAS_CREATOR]-(post:Post)
WHERE (post)-[:HAS_TAG]->(:Tag)<-[:HAS_INTEREST]-(person)
WITH friend, personCityName, allPostCount, count(post) AS commonPostCount
RETURN friend.id AS personId,
       friend.firstName AS personFirstName,
       friend.lastName AS personLastName,
       friend.gender AS personGender,
       personCityName,
  CASE allPostCount
    WHEN 0 THEN 0.0
    ELSE commonPostCount / (allPostCount + 0.0)
  END AS commonInterestScore
ORDER BY commonInterestScore DESC, personId ASC
LIMIT {limit}
```

A.3.11 Query 11

```
MATCH (:Person {id:{person_id}})-[:KNOWS*1..2]-(friend:Person)
WITH DISTINCT friend
MATCH (friend)-[worksAt:WORKS_AT]->(company:Company)
WHERE worksAt.workFrom <= {work_from_year} AND
     (company)-[:IS_LOCATED_IN]->(:Country {name:{country_name}})
RETURN friend.id AS friendId,
       friend.firstName AS friendFirstName,
       friend.lastName AS friendLastName,
       worksAt.workFrom AS workFromYear,
       company.name AS companyName
ORDER BY workFromYear ASC, friendId ASC
LIMIT {limit}
```

A.3.12 Query 12

```
MATCH (:Person {id:{person_id}})-[:KNOWS]-(friend:Person)
OPTIONAL MATCH (friend)<-[:HAS_CREATOR]-
               (comment:Comment)-[:REPLY_OF*]->
               ()-[:HAS_TAG]->(tag:Tag)-[:HAS_TYPE]->
               (tagClass:TagClass)-[:IS_SUBCLASS_OF*0..]->(baseTagClass:TagClass)
WHERE tagClass.uri = {tag_class_id} OR baseTagClass.uri = {tag_class_id}
RETURN friend.id AS friendId,
       friend.firstName AS friendFirstName,
       friend.lastName AS friendLastName,
       collect(DISTINCT tag.name) AS tagNames,
       count(DISTINCT comment) AS count
ORDER BY count DESC, friendId ASC
LIMIT {limit}
```

A.3.13 Query 13

```
MATCH path = shortestPath((person1:Person {id:{person_id_1}})-[:KNOWS]-
                          (person2:Person {id:{person_id_2}}))
RETURN length(path) AS pathLength
```

A.3.14 Query 14

```
MATCH path = (person1:Person {id:{person_id_1}})<-[:HAS_CREATOR]-
             ()-[r:REPLY_OF*0..]-
             ()-[:HAS_CREATOR]->(person2:Person {id:{person_id_2}})
WHERE all(message IN [n IN nodes(path)
WHERE not(n:Person)]
WHERE (message)-[:HAS_CREATOR]->(person1) OR (message)-[:HAS_CREATOR]->(person2))
RETURN
  [n IN nodes(path) | [labels(n)[0], n.id]] AS pathNodes,
  reduce(weight = -0.5, n IN nodes(path) |
    CASE labels(n)[0]
      WHEN 'Post' THEN weight + 1.0
      WHEN 'Comment' THEN weight + 0.5
      ELSE weight
    END) AS weight
ORDER BY length(pathNodes) ASC, weight DESC
LIMIT {limit}
```

B Scale Factor Statistics

B.1 Scale Factor Statistics

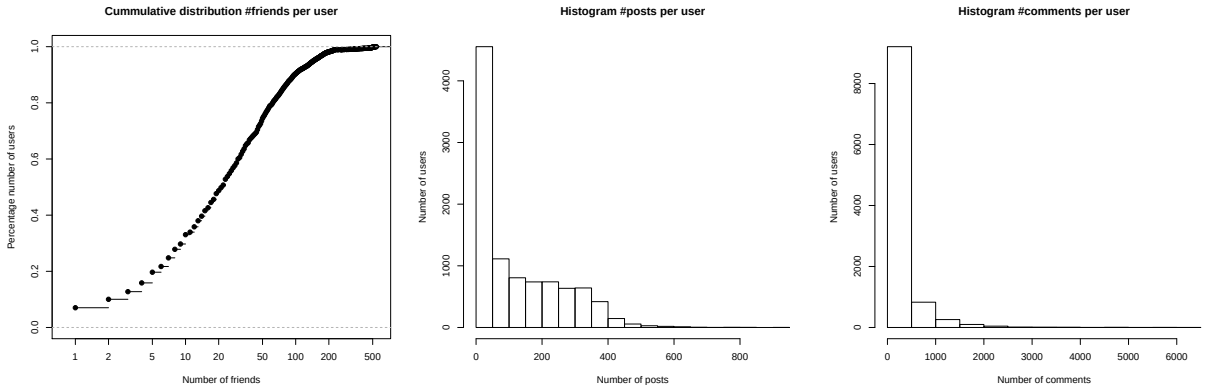
B.1.1 Scale Factor 1

Entity	Num Entities	Bytes
comment	2343952	254723836
forum	110202	6548409
organisation	7996	813270
person	11000	990357
place	1466	83667
post	1214766	138430549
tag	16080	1122429
tagclass	71	3946
Relation	**Num Relations**	**Bytes**
comment_hasCreator_person	2343952	63507355
comment_hasTag_tag	3069162	57501504
comment_isLocatedIn_place	2343952	39543099
comment_replyOf_comment	1187815	31674987
comment_replyOf_post	1156137	30828349
forum_containerOf_post	1214766	32211087
forum_hasMember_person	3260578	159205747
forum_hasModerator_person	110202	3017841
forum_hasTag_tag	355354	6527532
organisation_isLocatedIn_place	7996	79310
person_isLocatedIn_place	11000	196342
person_hasInterest_tag	256152	5120644
person_knows_person	452622	22659548
person_likes_comment	1649394	80566053
person_likes_post	1170372	57185940
person_studyAt_organisation	8820	221093
person_workAt_organisation	23969	581247
place_isPartOf_place	1460	11965
post_hasCreator_person	1214766	33212920
post_hasTag_tag	789735	14621607
post_isLocatedIn_place	1214766	20529353
tag_hasType_tagclass	16080	163348
tagclass_isSubclassOf_tagclass	70	616
Property Files	**Num Properties**	**Bytes**
person_email_emailaddress	18602	831575
person_speaks_language	24204	437214
Total Entities	**Total Relations**	**Total Bytes**
3705533	21859120	1063152739

Table B.1: General statistics for SF 1

SF = 1				
Clustering Coef.	0.0484			
	Min	**Max**	**Mean**	**Median**
#comments/user	1	6002	224	82
#posts/user	1	912	123	66
#friends/user	1	540	41	22
#likes/user	1	2725	260	171

Table B.2: Detail statistics for SF 1



(a) Cumulative number of friends. (b) Histogram #posts per user. (c) Histogram #comments per user.

Figure B.1: Data distributions for SF 1

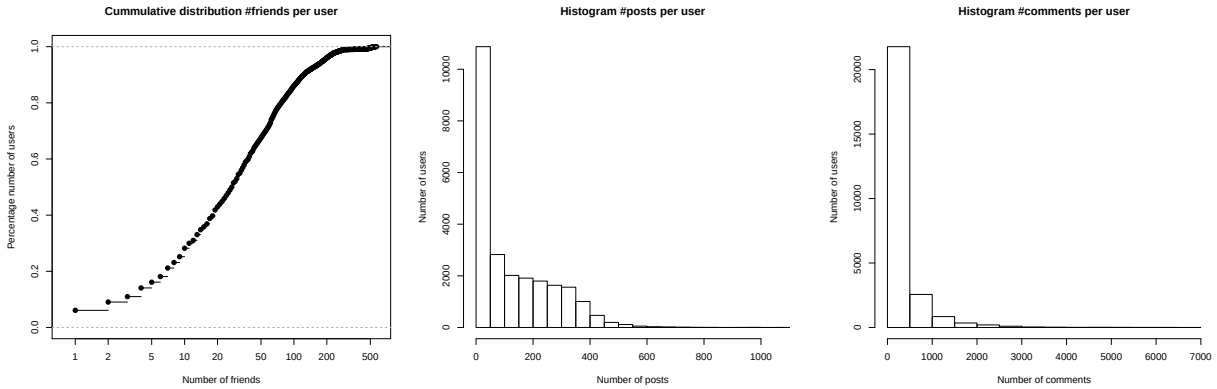
B.1.2 Scale Factor 3

Entity	Num Entities	Bytes
comment	7135636	776534811
forum	272268	16231309
organisation	7996	813270
person	27000	2431528
place	1466	83667
post	3140119	374416646
tag	16080	1122429
tagclass	71	3946
Relation	**Num Relations**	**Bytes**
comment_hasCreator_person	7135636	194770123
comment_hasTag_tag	9264389	174656230
comment_isLocatedIn_place	7135636	121173303
comment_replyOf_comment	3619711	97338366
comment_replyOf_post	3515925	94545033
forum_containerOf_post	3140119	83915474
forum_hasMember_person	9939453	486936117
forum_hasModerator_person	272268	7495375
forum_hasTag_tag	873831	16205018
organisation_isLocatedIn_place	7996	79310
person_isLocatedIn_place	27000	482925
person_hasInterest_tag	628563	12575921
person_knows_person	1370174	68746822
person_likes_comment	5555074	272259351
person_likes_post	3629288	177882573
person_studyAt_organisation	21574	541636
person_workAt_organisation	58843	1428856
place_isPartOf_place	1460	11965
post_hasCreator_person	3140119	86258384
post_hasTag_tag	2384629	44446829
post_isLocatedIn_place	3140119	53352987
tag_hasType_tagclass	16080	163348
tagclass_isSubclassOf_tagclass	70	616
Property Files	**Num Properties**	**Bytes**
person_email_emailaddress	45573	2041123
person_speaks_language	59467	1076428
Total Entities	**Total Relations**	**Total Bytes**
10600636	64877957	3170021719

Table B.3: General statistics for SF 3

SF = 3				
Clustering Coef.	0.0456			
	Min	**Max**	**Mean**	**Median**
#comments/user	1	6631	275	102
#posts/user	1	1096	128	72
#friends/user	1	569	51	28
#likes/user	1	3057	344	231

Table B.4: Detail statistics for SF 3



(a) Cumulative number of friends. (b) Histogram #posts per user. (c) Histogram #comments per user.

Figure B.2: Data distributions for SF 3

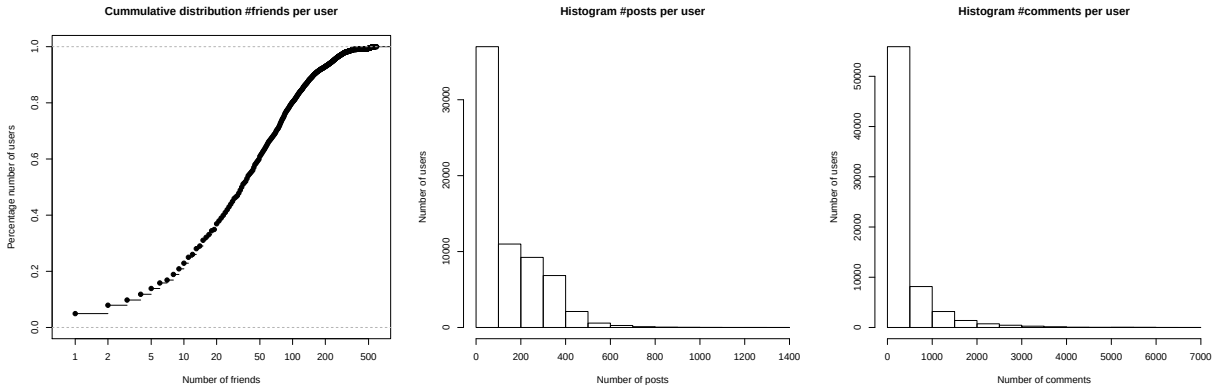
B.1.3 Scale Factor 10

Entity	Num Entities	Bytes
comment	24271888	2648214861
forum	729153	43643724
organisation	7996	813324
person	73000	6570890
place	1466	83721
post	8915649	1126585578
tag	16080	1122468
tagclass	71	3985
Relation	**Num Relations**	**Bytes**
comment_hasCreator_person	24271888	669164047
comment_hasTag_tag	31753457	605414570
comment_isLocatedIn_place	24271888	418145702
comment_replyOf_comment	12306670	336987410
comment_replyOf_post	11965218	327636871
forum_containerOf_post	8915649	242973393
forum_hasMember_person	33883607	1670125108
forum_hasModerator_person	729153	20284418
forum_hasTag_tag	2369727	44544367
organisation_isLocatedIn_place	7996	79388
person_isLocatedIn_place	73000	1305804
person_hasInterest_tag	1713574	34283207
person_knows_person	4654416	233569942
person_likes_comment	21418614	1054924693
person_likes_post	12661782	623979230
person_studyAt_organisation	58429	1467151
person_workAt_organisation	158961	3860488
place_isPartOf_place	1460	12022
post_hasCreator_person	8915649	247527557
post_hasTag_tag	8216364	154770790
post_isLocatedIn_place	8915649	154055825
tag_hasType_tagclass	16080	163408
tagclass_isSubclassOf_tagclass	70	691
Property Files	**Num Properties**	**Bytes**
person_email_emailaddress	124555	5574325
person_speaks_language	160779	2910238
Total Entities	**Total Relations**	**Total Bytes**
34015303	217279301	10680799196

Table B.5: General statistics for SF 10

SF = 3				
Clustering Coef.	0.0456			
	Min	**Max**	**Mean**	**Median**
#comments/user	1	6631	275	102
#posts/user	1	1096	128	72
#friends/user	1	569	51	28
#likes/user	1	3057	344	231

Table B.6: Detail statistics for SF 10



(a) Cumulative number of friends. (b) Histogram #posts per user. (c) Histogram #comments per user.

Figure B.3: Data distributions for SF 10

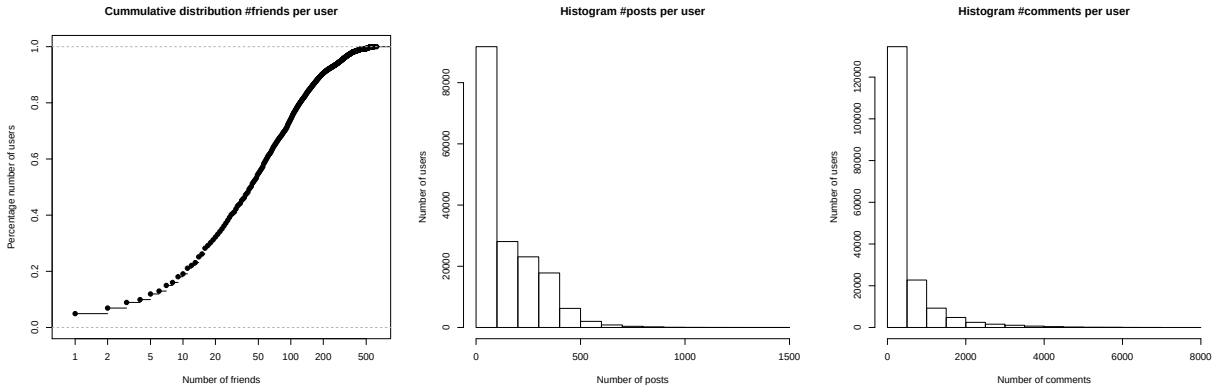
B.1.4 Scale Factor 30

Entity	Num Entities	Bytes
comment	73590941	8083989095
forum	1842141	111539981
organisation	7996	813396
person	184000	16572878
place	1466	83793
post	23765756	3155561666
tag	16080	1122520
tagclass	71	4037
Relation	**Num Relations**	**Bytes**
comment_hasCreator_person	73590941	2088295129
comment_hasTag_tag	96053813	1903298754
comment_isLocatedIn_place	73590941	1320854361
comment_replyOf_comment	37324357	1075860096
comment_replyOf_post	36266584	1045376200
forum_containerOf_post	23765756	679608557
forum_hasMember_person	103901443	5196088120
forum_hasModerator_person	1842141	52580681
forum_hasTag_tag	5976729	116509043
organisation_isLocatedIn_place	7996	79492
person_isLocatedIn_place	184000	3297409
person_hasInterest_tag	4318588	86533802
person_knows_person	14212356	714378938
person_likes_comment	71641419	3584484467
person_likes_post	39694513	1986127459
person_studyAt_organisation	147005	3695367
person_workAt_organisation	401356	9761198
place_isPartOf_place	1460	12098
post_hasCreator_person	23765756	677464115
post_hasTag_tag	24931521	488840146
post_isLocatedIn_place	23765756	426900332
tag_hasType_tagclass	16080	163488
tagclass_isSubclassOf_tagclass	70	791
Property Files	**Num Properties**	**Bytes**
person_email_emailaddress	312925	14030700
person_speaks_language	405403	7353001
Total Entities	**Total Relations**	**Total Bytes**
99408451	655400581	32851281110

Table B.7: General statistics for SF 30

SF = 30				
Clustering Coef.	0.0439			
	Min	**Max**	**Mean**	**Median**
#comments/user	1	7592	413	155
#posts/user	1	1412	139	83
#friends/user	1	625	77	43
#likes/user	1	3828	610	420

Table B.8: Detail statistics for SF 30



(a) Cumulative number of friends. (b) Histogram #posts per user. (c) Histogram #comments per user.

Figure B.4: Data distributions for SF 30

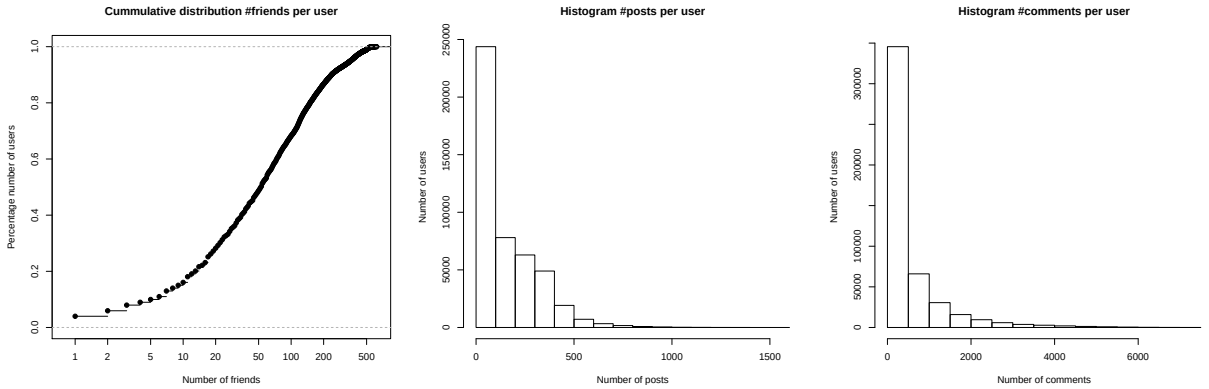
B.1.5 Scale Factor 100

Entity	Num Entities	Bytes
comment	243266898	26732787716
forum	5002291	303107584
organisation	7996	813396
person	499000	44950237
place	1466	83793
post	68871360	9601082178
tag	16080	1122520
tagclass	71	4037
Relation	**Num Relations**	**Bytes**
comment_hasCreator_person	243266898	6923334782
comment_hasTag_tag	317369562	6310390486
comment_isLocatedIn_place	243266898	4380836100
comment_replyOf_comment	123386519	3571363911
comment_replyOf_post	119880379	3469854233
forum_containerOf_post	68871360	1977411509
forum_hasMember_person	341232279	17085982726
forum_hasModerator_person	5002291	143155976
forum_hasTag_tag	16195463	317441296
organisation_isLocatedIn_place	7996	79492
person_isLocatedIn_place	499000	8948068
person_hasInterest_tag	11692172	234436590
person_knows_person	46598276	2343165388
person_likes_comment	260701994	13062653343
person_likes_post	135205141	6773886764
person_studyAt_organisation	398560	10023920
person_workAt_organisation	1086037	26420132
place_isPartOf_place	1460	12098
post_hasCreator_person	68871360	1968125668
post_hasTag_tag	82466083	1623280287
post_isLocatedIn_place	68871360	1240297918
tag_hasType_tagclass	16080	163488
tagclass_isSubclassOf_tagclass	70	791
Property Files	**Num Properties**	**Bytes**
person_email_emailaddress	850804	38160557
person_speaks_language	1099440	19951911
Total Entities	**Total Relations**	**Total Bytes**
317665162	2154887238	108213328895

Table B.9: General statistics for SF 100

SF = 100				
Clustering Coef.	0.0422			
	Min	**Max**	**Mean**	**Median**
#comments/user	1	7465	502	190
#posts/user	1	1509	148	90
#friends/user	1	619	93	53
#likes/user	1	4312	799	556

Table B.10: Detail statistics for SF 100



(a) Cumulative number of friends. (b) Histogram #posts per user. (c) Histogram #comments per user.

Figure B.5: Data distributions for SF 100

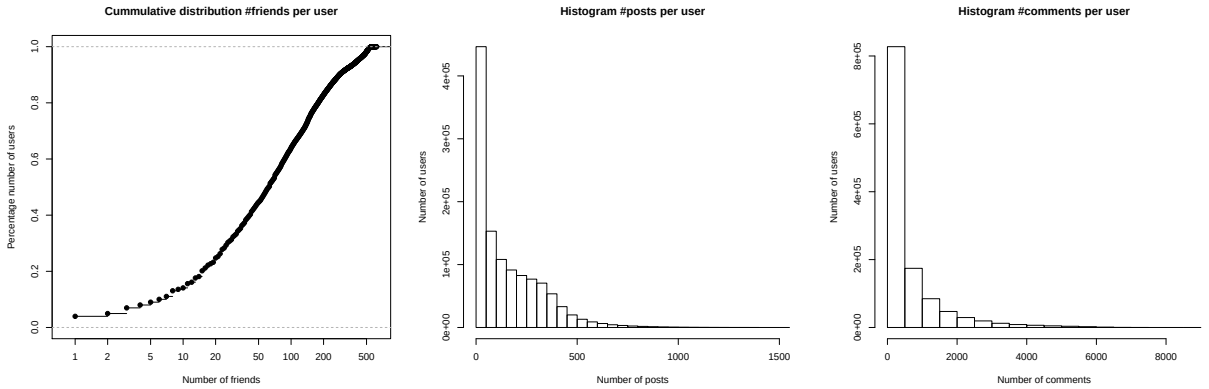
B.1.6 Scale Factor 300

Entity	Num Entities	Bytes
comment	710752235	78578510866
forum	12561079	769736017
organisation	7996	813396
person	1254000	113011768
place	1466	83793
post	182980982	26615002745
tag	16080	1122520
tagclass	71	4037
Relation	**Num Relations**	**Bytes**
comment_hasCreator_person	710752235	20740234727
comment_hasTag_tag	926124724	19010889474
comment_isLocatedIn_place	710752235	13268389734
comment_replyOf_comment	360517003	10910496465
comment_replyOf_post	350235232	10599470746
forum_containerOf_post	182980982	5473898610
forum_hasMember_person	995330706	50531158002
forum_hasModerator_person	12561079	368655691
forum_hasTag_tag	40653342	819806778
organisation_isLocatedIn_place	7996	79492
person_isLocatedIn_place	1254000	22520270
person_hasInterest_tag	29346263	589162363
person_knows_person	136219368	6857187354
person_likes_comment	820056009	41645511118
person_likes_post	404808353	20560829944
person_studyAt_organisation	1002380	25237062
person_workAt_organisation	2728559	66447988
place_isPartOf_place	1460	12098
post_hasCreator_person	182980982	5363521573
post_hasTag_tag	241151541	4898991661
post_isLocatedIn_place	182980982	3419470093
tag_hasType_tagclass	16080	163488
tagclass_isSubclassOf_tagclass	70	791
Property Files	**Num Properties**	**Bytes**
person_email_emailaddress	2140338	96111129
person_speaks_language	2763075	50221509
Total Entities	**Total Relations**	**Total Bytes**
907573909	6292461581	321396753302

Table B.11: General statistics for SF 300

SF = 300				
Clustering Coef.	0.0411			
	Min	**Max**	**Mean**	**Median**
#comments/user	1	8806	582	224
#posts/user	1	1501	155	97
#friends/user	1	620	109	63
#likes/user	1	4686	983	689

Table B.12: Detail statistics for SF 300



(a) Cumulative number of friends. (b) Histogram #posts per user. (c) Histogram #comments per user.

Figure B.6: Data distributions for SF 300

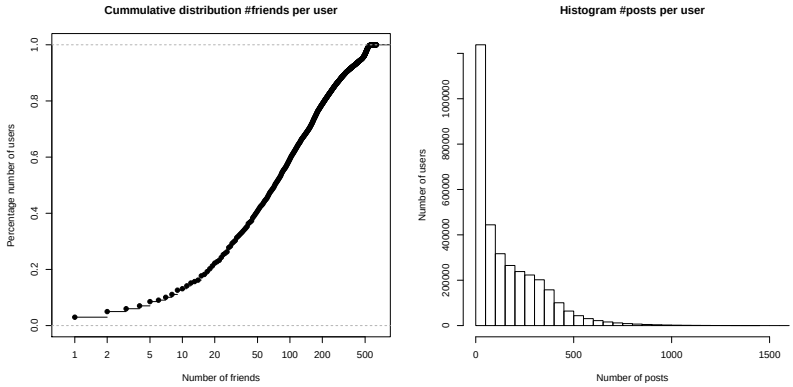
B.1.7 Scale Factor 1000

Entity	Num Entities	Bytes
comment	2335637135	258944003306
forum	36098481	2222966076
organisation	7996	813396
person	3600000	324485964
place	1466	83793
post	555306166	83647390485
tag	16080	1122520
tagclass	71	4037
Relation	**Num Relations**	**Bytes**
comment_hasCreator_person	2335637135	69009917568
comment_hasTag_tag	3042978961	63451008509
comment_isLocatedIn_place	2335637135	44333145872
comment_replyOf_comment	1184778982	36597884006
comment_replyOf_post	1150858153	35549852967
forum_containerOf_post	555306166	16985930071
forum_hasMember_person	3277239057	167465482785
forum_hasModerator_person	36098481	1071895282
forum_hasTag_tag	116727525	2398752244
organisation_isLocatedIn_place	7996	79492
person_isLocatedIn_place	3600000	64736060
person_hasInterest_tag	84229044	1692899009
person_knows_person	447163916	22530441760
person_likes_comment	2858070323	146129764930
person_likes_post	1361722197	69623238723
person_studyAt_organisation	2878718	72544726
person_workAt_organisation	7829672	190876421
place_isPartOf_place	1460	12098
post_hasCreator_person	555306166	16467663132
post_hasTag_tag	793254841	16381717061
post_isLocatedIn_place	555306166	10543790321
tag_hasType_tagclass	16080	163488
tagclass_isSubclassOf_tagclass	70	791
Property Files	**Num Properties**	**Bytes**
person_email_emailaddress	6141306	276081939
person_speaks_language	7932926	144358836
Total Entities	**Total Relations**	**Total Bytes**
2930667395	20704648244	1066123107668

Table B.13: General statistics for SF 1000

SF = 1000				
	Min	Max	Mean	Median
#posts/user	1	1576	163	103
#friends/user	1	636	124	73

Table B.14: Detail statistics for SF 1000



(a) Cumulative number of friends. (b) Histogram #posts per user.

Figure B.7: Data distributions for SF 1000