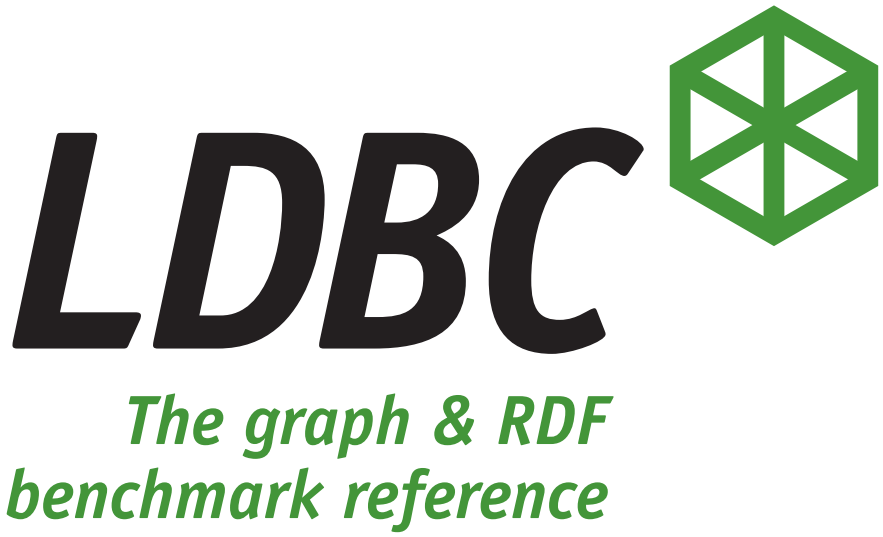


The LDBC Social Network Benchmark (version 0.4.0-SNAPSHOT (8ee86ec))

The specification was built on the source code available at

`https://github.com/ldbc/ldbc_snb_docs/tree/dev`

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, LDBC SNB uses the recognizable scenario of operating a social network, characterized by its graph-shaped data.

LDBC SNB consists of two workloads that focus on different functionalities: the Interactive workload (interactive transactional queries) and the Business Intelligence workload (analytical queries).

This document contains the definition of the Interactive Workload and the first draft of the Business Intelligence Workload. This includes a detailed explanation of the data used in the LDBC SNB benchmark, a detailed description for all queries, and instructions on how to generate the data and run the benchmark with the provided software.

Executive Summary

The new data economy era, based on complexly structured, distributed and large datasets, has brought on new demands on data management and analytics. As a consequence, new industry actors have appeared, offering technologies specially built for the management of graph-like data. Also, traditional database technologies, such as relational databases, are being adapted to the new demands to remain competitive.

LDBC's Social Network Benchmark (LDBC SNB) is an industrial and academic initiative, formed by principal actors in the field of graph-like data management. Its goal is to define a framework where different graph based technologies can be fairly tested and compared, that can drive the identification of systems' bottlenecks and required functionalities, and can help researchers to open new research frontiers.

The philosophy around which LDBC SNB is designed is to be easy to understand, flexible and cheap to adopt. For all these reasons, LDBC SNB will propose different workloads representing all the usage scenarios of graph-like database technologies, hence, targeting systems of different nature and characteristics. In order increase its adoption by industry and research institutions, LDBC SNB provides all necessary software, which are designed to be easy to use and deploy at a small cost.

This document contains:

- A detailed specification of the data used in the whole LDBC SNB benchmark.
- A detailed specification of the workloads.
- A detailed specification of the execution rules of the benchmark.
- A detailed specification of the auditing rules and the full disclosure report's required contents.

Table of Contents

Acknowledgments

Special thanks to all the people that have contributed to the development of this benchmark:

- Renzo Angles (Universidad de Talca)
- Alex Averbuch (Neo4j)
- Peter Boncz (Vrije Universiteit Amsterdam and CWI)
- Márton Búr (McGill University)
- Orri Erling (OpenLink Software)
- Andrey Gubichev (Technische Universität München)
- Moritz Kaufmann (Technische Universität München)
- Josep Lluís Larriba Pey (Universitat Politècnica de Catalunya)
- Minh-Duc Pham (Altran)
- Marcus Paradies (SAP and DLR)
- Arnau Prat-Pérez (DAMA UPC and Sparsity Technologies)
- Mirko Spasić (OpenLink Software)
- Norbert Martínez (Huawei Technologies)
- Gábor Szárnyas (MTA-BME Lendület Research Group on Cyber-Physical Systems and Budapest University of Technology and Economics [2016–2020], CWI [2020–])
- József Marton (Budapest University of Technology and Economics)
- János Benjamin Antal (Budapest University of Technology and Economics)
- Vlad Haprian (Oracle Labs)
- Benjamin A. Steer (Queen Mary University of London)
- Jack Waudby (Newcastle University)

Definitions

This section defines fundamental concepts used in the LDBC benchmark terminology. Part of the definitions below are repeated from the LDBC benchmark specification document.

LDBC SNB The Linked Data Benchmark Council's Social Network Benchmark suite which currently consists of the Interactive workload and a preliminary version of the Business Intelligence workload.

System Under Test (SUT) This is the totality of the hardware and software that participates in a benchmark run, excluding parts that are exclusively used for driving the workload. If the parts driving the workload are collocated on the same operating system instance as the SUT, then this is also considered a part of the SUT. In client-server configurations where the test driver is not on a machine hosting any DBMS function the SUT is not considered to encompass the hardware or software which exclusively serves to drive the test workload.

Datagen This module is provided by LDBC SNB and produces the standard benchmark datasets to be loaded into the SUT for the benchmark. The data generation phase is not part of running the benchmark.

Test Driver (Benchmark Driver, Driver) The test driver refers to the parts of the benchmark run that coordinate query execution and, if prescribed by a given benchmark, data loading.

Workload (Benchmark) This is the totality of the tasks a particular benchmark performs against an SUT. This includes data loading as well as the query/update workload. This does not include preparatory stages such as generating benchmark data with a data generator or transferring the data to the platform constituting the SUT. The terms workload and benchmark are synonyms in this context.

Time Compression Ratio (TCR) This parameter compresses (or stretches) durations between operation start times to increase (or decrease) operation rate, thereby allowing systems to reach their maximum throughput for a given workload. The smaller this number is, the higher compression ratio it represents (e.g. 2.0 means run benchmark 2x slower, while 0.1 = run benchmark 10x faster). Systems are expected to compete on achieving high $\text{TCR}^{-1} = \frac{1}{\text{TCR}}$.

Query mix The ratio of read and update queries of a workload, and the frequency at which they are issued.

Scale Factor (SF) 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. The scale factor refers to the measured size of the data in Gigabytes when serialised in CsvSingularProjectedFK.

Validation Step The benchmark specifies a scale factor for which ACID test cases are executed and the query results are compared to a reference result set (i.e. expected output). This step is required to use the very same set of queries and data structures (this includes both PDS, IADS and EADS – defined below) that are used in the actual benchmark runs.

Schema (Database Schema) A schema is the totality of the non-built-in declarations which are fed into the SUT prior to running a workload. For a relational system, the schema consists of tables, indices, views, materialised views and declarative constraints (e.g. foreign key and not null constraints). An ontology for an RDF system counts as a schema if it is loaded on the SUT. An RDF SUT may have no schema at all and still run the workload. However, any declaration or setting (e.g. indices) that is not on by default in the SUT, but is used in at least one case of the benchmark run counts as part of the schema. The schema does not include stored procedures, triggers, or other imperative (procedural) application specific code that may reside on the SUT and could impact the benchmark results. The schema is required to be the same across all benchmark runs using the same scale factor for a given workload.

Primary Data Structure (PDS) This is anything that may influence the result of a database query or may be changed by an update of the database. These may be resident in RAM or durable media or both. Examples of data structures are database base tables and adjacency lists.

Implicit Auxiliary Data Structure (IADS) This is a data structure for providing more efficient access to all or parts of the primary data structure. IADS are created by the DBMS automatically and the system may allow them to be turned off.

> Some systems, such as many RDF stores have multiple covering indices on the primary data structure. The definition in this case is that the primary data structure consists of all the dif-

ferently ordered full copies of the base table; a table of subject predicate object graph (SPOG) in the RDF case. In this same instance, Auxiliary data structures comprise any data structure which materialise a subset of the SPOG.

Explicit Auxiliary Data Structure (EADS) These are any application or workload profile specific structures that are declared in addition to the PDSs and IADSs managed by the SUT. These duplicate the data and are created with explicit statements. Secondary indices, materialised views, with or without aggregates, are all examples of this in a relational context. The decision about the used EADS is always part of the schema declaration.

In the case of relational systems, an ADS may be an index from primary key values to a heap table, if the system in question has such concepts. A secondary index of a relational table, in its memory based and durable media based manifestations is an example for EADS. Such a secondary index is not considered an ADS since it must be declared, which makes its creation explicit. An ADS must be implicit and not created by any specific DDL statement or directive. In the case of RDF systems, if the implementation supports user definable index schemes, as long as these are defined once and apply to all triples/quads, such structures are designated as ADS. If an RDF system selectively makes data structures which apply to some quads but not to others, then such structures are designated as EADS.

SUT-Resident Logic This is any application specific code that is resident on the SUT, whether by static linking, dynamic loading, JIT, interpretation or any other means of embedding application specific logic into a generic DBMS. Examples of this are stored procedures, hosting Java, CLR or other run times in the SUT process (or processes), loading application specific libraries to extend native functions or data structures etc. A special case is that of a database exclusively accessed via an in-process API. In these cases, any code that is not the test driver or a workload implementation expressed against a generally supported API of the DBMS is deemed SUT resident logic in addition to any other code which may fit the above definitions.

Test Sponsor The party which initiates an audit of a benchmark implementation over an SUT. This is typically the vendor of a key component of the SUT, e.g. DBMS or hardware.

Full Disclosure Report (FDR) This is a document which allows reproduction of any audited benchmark result by a third party. It contains complete description of the circumstances of the benchmark run, including version and configuration of SUT, dataset and test driver.

1 Introduction

1.1 Motivation for the Benchmark

The new era of data economy, based on large, distributed, and complexly structured datasets, has brought on new and complex challenges in the field of data management and analytics. These datasets, usually modeled as large graphs, have attracted both industry and academia, due to 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 these new emerging technologies, as they set a framework where different systems can compete and be compared in a fair way, they let technology providers 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 a comprehensive benchmark by 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 cases of modern graph-like applications.

LDBC SNB includes the Interactive Workload [24], which consists of user-centric transactional-like interactive queries, and the Business Intelligence Workload, which includes analytic queries to respond to business-critical questions. Initially, a graph analytics workload was also included in the roadmap of LDBC SNB, but this was finally delegated to the Graphalytics benchmark project [33, 34], which was adopted as an official LDBC graph analytics benchmark. LDBC SNB and Graphalytics combined target a broad range of systems with different nature and characteristics. LDBC SNB and Graphalytics aim at capturing the essential features of these scenarios while abstracting away details of specific business deployments.

This document contains the definition of the Interactive Workload and the first draft of the Business Intelligence Workload. This includes a detailed explanation of the data used in the LDBC SNB benchmark, a detailed description for all queries, and instructions on how to generate the data and run the benchmark with the provided software.

1.2 Relevance to the 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 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/JanusGraph) are novel technologies aimed at storing directed and labeled graphs. They support graph traverals, typically by means of APIs, though some of them also support dedicated graph-oriented query languages (e.g. Neo4j's Cypher). These systems' internal structures are typically designed to store dynamic graphs that change over time. They offer support for transactional queries with some degree of consistency, and value-based indices to quickly

locate nodes and edges. Finally, their architecture is typically single-machine (non-cluster). These systems can potentially implement all three workloads, though Interactive and Business Intelligence workloads are where they will presumably be more competitive.

mention property graphs

- **Graph processing frameworks** (e.g. Giraph, Signal/Collect, GraphLab, Green Marl) are designed to perform global graph computations, executed in parallel or in a lockstep fashion. These computations are typically long latency, involving many nodes and edges and often consist of approximation answers to NP-complete problems. These systems expose an API, sometimes following a vertex-centric paradigm, and their architecture targets both single-machine and cluster systems. These systems will likely implement the Graph Analytics workload.
- **RDF database systems** (e.g. OWLIM, Virtuoso, BigData, Jena TDB, Stardog, AllegroGraph) are systems that implement the SPARQL 1.1 query language, similar in complexity to SQL-92, which allows for structured queries, and simple traversals. RDF database systems 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 and their supposed strength is integrating multiple data sources (e.g. DBpedia). Their architecture is both single-machine and clustered, and they will likely target Interactive and Business Intelligence workloads.
- **Relational database systems** (e.g. Postgres, MySQL, Oracle, IBM DB2, Microsoft SQL Server, Virtuoso, MonetDB, Vectorwise, Vertica, but also Hive and Impala) treat graph data relationally, and queries are formulated in SQL and/or PL/SQL. Both single-machine and cluster systems exist. They 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) are cluster-based and scalable. 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. MapReduce systems could be suited for the Graph Analytics workload, but their query latency would presumably be 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 Business Intelligence workload.

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, in 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 in accordance with the LDBC Byelaws [40]. 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 in 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 Chapter 3, LDBC SNB defines the schema of the data used in the benchmark. The schema represents a realistic social network, including people and their activities in the social network during a period of time. Personal information of each person, such as name, birthday, interests or places where people work or study, is included. A person's activity is represented in the form of friendship relationships and content sharing (i.e. messages and pictures). LDBC SNB provides a scalable synthetic data generator based on the MapReduce paradigm, which 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 Chapter 4, the Interactive Workload and the first draft of the Business Intelligence workload are proposed. 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. The Interactive workload reproduces the interaction between the users of the social network by including lookups and transactions, which update small portions of the database. These queries are designed to be interactive and target systems capable of responding to such queries with low latency for multiple concurrent users. The Business Intelligence workload represents analytic queries a social network company would like to perform in the social network, to take advantage of the data and to discover new business opportunities. This workload explores moderate to large portions of the graph from different entities, and performs more resource-intensive operations.

LDBC SNB provides an execution test driver, which is responsible for 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 automatically handles the validation of the queries by means of a validation dataset provided by LDBC. The overall philosophy of LDBC SNB is to provide the necessary software tools to run the benchmark, and therefore to reduce the benchmark's entry point as much as possible.

1.4 Related Projects

Along the Social Network Benchmark, LDBC [7] provides other benchmarks as well:

- The Semantic Publishing Benchmark (SPB) [60] measures the performance of *semantic databases* operating on RDF datasets.
- The Graphalytics benchmark [33] measures the performance of *graph analysis* operations (e.g. PageRank, local clustering coefficient).

1.5 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, which meets all the desired needs. The list of participants is the following:

- FOUNDATION FOR RESEARCH AND TECHNOLOGY HELLAS
- MTA-BME LENDUELET RESEARCH GROUP ON CYBER-PHYSICAL SYSTEMS
- NEO4J

- ONTOTEXT
- 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.

1.6 Software components

The source code of this specification and the software components discussed here are available open-source:

- Specification: `https://github.com/ldbc/ldbc_snb_docs`
- Interactive workload:
  - Datagen (Hadoop-based): `https://github.com/ldbc/ldbc_snb_datagen_hadoop`
  - Driver: `https://github.com/ldbc/ldbc_snb_driver`
  - Reference implementations (in Cypher and SQL): `https://github.com/ldbc/ldbc_snb_interactive`
- Business Intelligence (BI) workload:
  - Datagen (Spark-based): `https://github.com/ldbc/ldbc_snb_datagen_spark`
  - Driver and reference implementations (in Cypher and SQL): `https://github.com/ldbc/ldbc_snb_bi`

Repositories have two distinguished branches: `stable` contains the code for the v0.3 benchmark, while `dev` contains the code for the v0.4 (work-in-progress) benchmark.

1.7 Technical report

This technical report is available on arXiv [6] and is updated upon new releases of the SNB.

2 Benchmark Specification

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 for executing LDBC SNB are limited to pure software requirements to be able to run the tools. While the benchmark specification aims to be portable, the software provided by LDBC SNB have been developed and tested under Linux-based operating systems. The driver and the clients for the reference implementations were implemented in Java. The generator has two versions: the Hadoop-based one was written in Java, while the Spark-based one is written in a mix of Java and Scala.

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 relational 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.

3 Data sets and data generation

This chapter 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.

Warning. This chapter describes the latest variant of the data set. If you are looking for information on the Interactive workload, please also check Appendix D.

3.1 Data Types

Table 3.1 describes the different data types used in the benchmark.

Type	Description
ID	integer type with 64-bit precision. All IDs within a single entity type (e.g. Person, Message) are unique, but different entity types (e.g. a Forum and a Tag) might have the same ID.
32-bit Integer	integer type with 32-bit precision
64-bit Integer	integer type with 64-bit precision
32-bit Float	integer type with 32-bit precision
64-bit Float	integer type with 64-bit precision
String	variable length text of size 40 Unicode characters
Long String	variable length text of size 256 Unicode characters
Text	variable length text of size 2000 Unicode characters
Date	date with a precision of a day, encoded as a string with the following format: `yyyy-mm-dd`, where `yyyy` is a four-digit integer representing the year, the year, `mm` is a two-digit integer representing the month and `dd` is a two-digit integer representing the day.
DateTime	date with a precision of milliseconds, encoded as a string with the following format: `yyyy-mm-ddTHH:MM:ss.sss+00:00`, where `yyyy` is a four-digit integer representing the year, the year, `mm` is a two-digit integer representing the month and `dd` is a two-digit integer representing the day, `HH` is a two-digit integer representing the hour, `MM` is a two digit integer representing the minute and `ss.sss` is a five digit fixed point real number representing the seconds up to millisecond precision. Finally, the `+00:00` of the end represents the timezone, which in this case is always GMT.
Boolean	logical type, taking the value of either True of False

Table 3.1: Description of the data types. Some types such as 32-bit Float and 64-bit Integer are currently not used in the benchmark.

3.2 Data Schema

Figure 3.1 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, Organisations, 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[1]. An example graph conforming the SNB schema is shown in Appendix E.

[1]Tags are basically equivalent to *hashtags* on contemporary social media sites. In this document, we occasionally use the term *topic* to refer to 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 Datagen component supports multiple output data formats to fit the needs of different types of systems, including RDF, relational DBMS and graph DBMS.

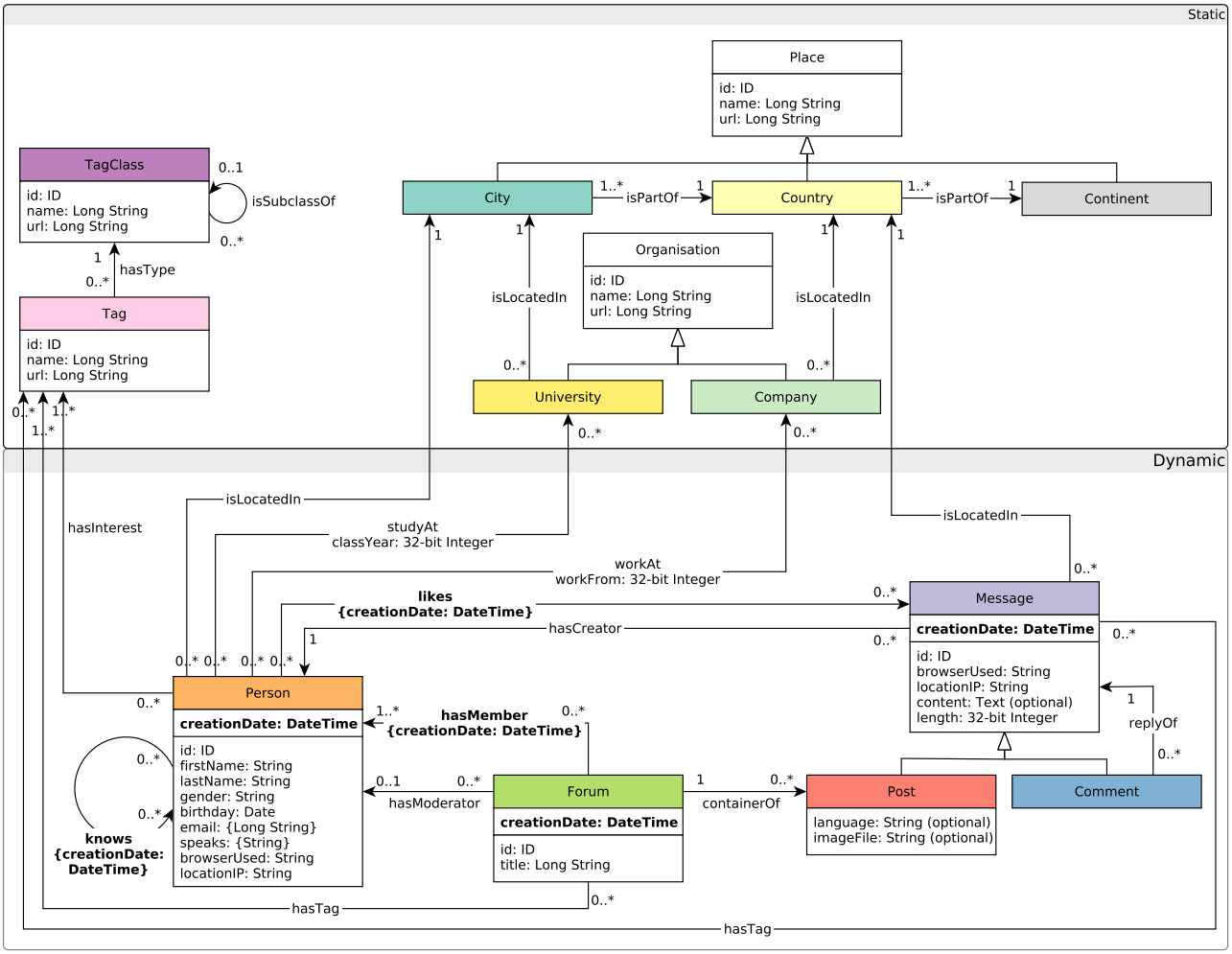


Figure 3.1: UML class diagram-style depiction of the LDBC SNB graph schema. Note that the knows edges should be treated as undirected (but are serialized only in a single direction). The cardinality of the hasModerator edge has changed between version 0.3.x (where it was exactly 1) and version 0.4.x (where it is 0..1).

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

Textual Restrictions and Notes

- Messageas always have a non-empty `content` attribute.
- Posts have either a `content` or an `imageFile` attribute (i.e. they always have exactly one of them.) The one they do not have is represented with an empty string or with `NULL`.
- Posts in a forum can be created by a non-member person if and only if that person is the moderator of the Forum.

3.2.1 Entities (nodes)

City: 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: a sub-class of a Message, and represents a comment made by a person to an existing message (either a Post or a Comment).

Company: a sub-class of an Organisation, and represents a company where persons work.

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

Country: a sub-class of a Place, and represents a country of the real world. Countries are selected as the place of operation for Companies as well as the location of Messages.

Forum: 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 it is not evident, there exist three different types of forums. They are distinguished by their titles:

- personal walls: "Wall of ..."
- image albums: "Album k of ..."
- groups: "Group for ..."

Table 3.2 shows the attributes of the Forum entity.

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

Table 3.2: Attributes of the Forum entity.

Message: an abstract entity that represents a message created by a person. Table 3.3 shows the attributes of the 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.
`content`	Text (optional)	The content of the message.
`length`	32-bit Integer	The length of the content.

Table 3.3: Attributes of the Message interface.

Organisation: an institution of the real world. Table 3.4 shows the attributes of the Organisation entity.

Attribute	Type	Description
`id`	ID	The identifier of the organisation.
`name`	Long String	The name of the organisation.
`url`	Long String	The URL of the organisation.

Table 3.4: Attributes of the Organisation entity.

Person: the avatar a real world person creates when he/she joins the network, and contains various information about the person as well as network related information. Table 3.5 shows the attributes of the 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	{Long String}	The set of emails the person has (cardinality: at least one).
speaks	{String}	The set of languages the person speaks (cardinality: at least one).
browserUsed	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 3.5: Attributes of the Person entity.

Place: a place in the world. Table 3.6 shows the attributes of the Place entity. Note, each Place has additional parameters: longitude and latitude, which are not exposed. These are used internally for sorting places.

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

Table 3.6: Attributes of the Place entity.

Post: 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 content or imageFile, always one of them but never both. The one they do not have is an empty string. Table 3.7 shows the attributes of the Post entity.

Attribute	Type	Description
language	String (optional)	The language of the post. Mutually exclusive with `imageFile`.
imageFile	String (optional)	The image file of the post. Mutually exclusive with `language`.

Table 3.7: Attributes of the Post entity.

Tag: 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 3.8 shows the atltributes of the Tag entity.

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

Table 3.8: Attributes of the Tag entity.

TagClass: a class used to build a hierarchy of tags. Table 3.9 shows the attributes of the TagClass entity.

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

Table 3.9: Attributes of the TagClass entity.

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

3.2.2 Relations (edges)

Relations (edges) connect entities of different types. The endpoint entities are defined by their "id" attribute. Edge instances starting from or ending in a given node are treated as a set, i.e. no ordering is defined on the edges. Multiple edges (i.e. edges of the same type between two entity instances) are not allowed in SNB graphs.

Name	Source	Destination	Type	Description
containerOf	Forum[1]	Post[0..*]	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[1..*]	D	A Person and a Tag representing a topic the person is interested in
hasMember	Forum[0..*]	Person[1..*]	D	A Forum and its member (Person) In version 0.3.x: **Attribute**: joinDate **Type**: DateTime **Description**: The Date the person joined the Forum In version 0.4.0+: **Attribute**: creationDate **Type**: DateTime **Description**: The Date the person joined the Forum
hasModerator	Forum[0..*]	In version 0.3.x: Person[1] In version 0.4.0+: Person[0..1]	D	A Forum and its moderator (Person)
hasTag	Message[0..*]	Tag[0..*]	D	A Message and a Tag representing the message's topic
hasTag	Forum[0..*]	Tag[1..*]	D	A Forum and a Tag representing the forum's topic
hasType	Tag[0..*]	TagClass[1]	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 their 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..1]	D	A TagClass and its parent TagClass

knows	Person[0..*]	Person[0..*]	U	Two Persons that know each other. Note that (1) the knows edges are undirected (all other edge types are directed and (2) to avoid duplications, these edges are only serialized to one direction and it is the responsibility of the loader/implementation component to treat them as undirected. **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 3.10: Description of the data relations. Type – D: directed edge, U: undirected edge.

3.2.3 Domain Concepts

A *thread* consists of Messages, starting with a single Post and the Comments that – either directly or transitively – reply to that Post.

3.3 Data Generation

LDBC SNB provides Datagen (Data Generator), which produces synthetic datasets following the schema 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. Datagen is defined by the following characteristics:

- **Realism.** Data generated by Datagen mimics the characteristics of those found in a real social network. In Datagen, 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, Datagen is capable of generating datasets of different sizes, from a few Gigabytes to Terabytes. Datagen 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.** Datagen 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 benchmarks' results reproducible.
- **Usability.** LDBC SNB is designed to have an affordable entry point. As such, Datagen's design is severely influenced by this philosophy, and therefore it is designed to be as easy to use as possible.

3.3.1 Resource Files

Datagen uses a set of resource files with data extracted from DBpedia. Conceptually, Datagen 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 birthday 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. Datagen 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 3.11.

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 3.11: Resource files.

3.3.2 Graph Generation

Figure 3.2 conceptually depicts the full data generation process. The first step loads all the dictionaries and resource files, and initializes the Datagen parameters. Second, it generates all the Persons in the graph, and the minimum necessary information to operate. Part of this 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 [66].

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 [45, 14], and implies the presence of a larger amount of triangles than that expected in a random network. In order to reproduce this characteristic, Datagen 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 p 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 Datagen to reproduce correlations.

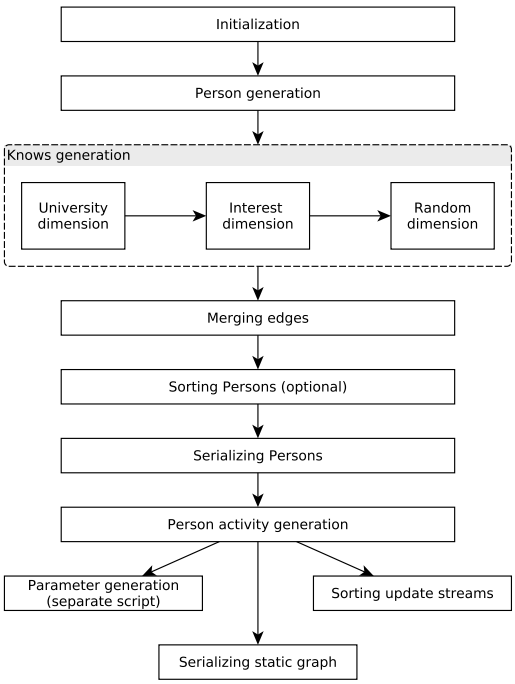


Figure 3.2: The Datagen generation process.

Given a similarity function $M(p) : p \rightarrow [0, \infty)$ 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 p against only the K neighbouring 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, Datagen uses a geometric probability distribution that provides a probability for picking persons to connect, that are between 1 and K 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 p, 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 Datagen, 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, Datagen 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, a person's activity, in the form of forums, posts and comments is created. Datagen 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 persons' activity in social network, are time correlations. Usually, person' posts creation in a social network is driven by real world events. For instance, one may think 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. Datagen 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 [41]. Furthermore, in order to reproduce the more uniform every day person activity, Datagen also generates posts uniformly distributed along all the simulated time.

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

3.3.3 Distributions, Parameters and Quirks

Interesting quirks:

- A Person is not a member of their Wall but they are its moderator, they do not have a hasMember edge.
- Each Album generated for Person will have approximately 70% of their friends as members.
- A given Person has a 5% chance of being a moderator of a set of groups.
- Group membership is composed of 30% from the moderator's friends and the remainder is chosen randomly (from the block the person is in).

  what is a block? explain
- Comments are only made in Walls and Groups.
- Messages can only receive likes during a 7-day window after their creation.
- Comments always occur within one day of Message they are replying to. The creation date is generated following the power-law distribution in Figure 3.3. The mean delay between Comments and their parent Message is 6.85 hours.
- Flashmob events span a 72-hour time window with a specific event time in the middle of the window; there are 36 hours each side of the specific event time. Following the distribution in Figure 3.4 posts are generated either side of flashmob event time, posts are clustered around the specific event time.

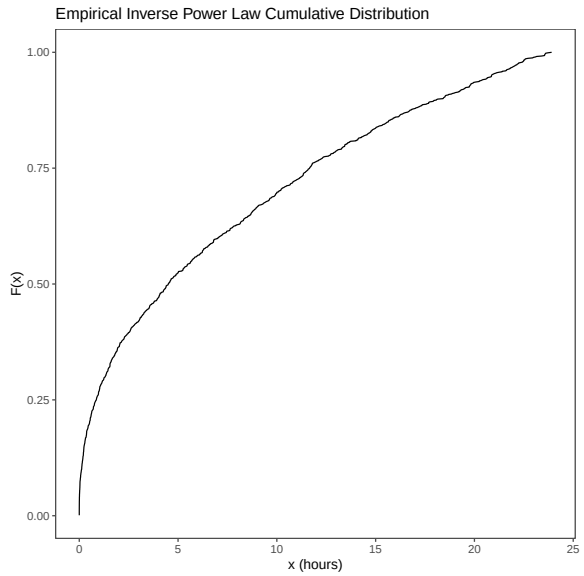


Figure 3.3: The power-law used to generate comments.

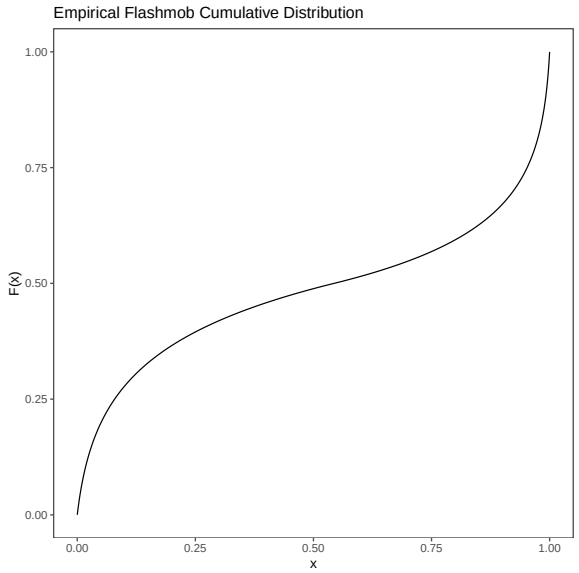


Figure 3.4: The distribution used to generate posts during flashmob events.

3.3.4 Implementation Details

Datagen 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 Datagen, 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 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.

3.4 Output Data

For each scale factor, Datagen produces three different artefacts:

- **Dataset:** The dataset to be bulk loaded by the SUT. In the Interactive workload, it corresponds to roughly the 90% of the total generated network.
- **Update Streams:** A set of update streams containing update queries, which are used by the driver to generate the update queries of the workloads. This update streams correspond to the remaining 10% of the generated dataset.
- **Substitution Parameters:** A set of files containing the different parameter bindings that will be used by the driver to generate the read queries of the workloads.

3.4.1 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 Gibibytes of the generated output files using the csv-singular-merged-fk serializer (see Section D.1.2) and default settings, i.e. both the 90% initial data and the 10% update streams count

Serializer name (v0.4+)	Legacy serializer name (v0.3)	Nodes	Attributes		Edges	
			single-valued	multi-valued	one-to-many	many-to-many
csv-singular-projected-fk	CsvBasic	⊗	○	⊗	⊗	⊗
csv-composite-projected-fk	CsvComposite	⊗	○	○	⊗	⊗
csv-singular-merged-fk	CsvMergeForeign	⊗	○	⊗	○	⊗
csv-composite-merged-fk	CsvCompositeMergeForeign	⊗	○	○	○	⊗

Table 3.13: Attributes and edges serialized to separate files the different CSV serializers.

towards the total size.[2] For example, SF1 takes roughly 1 GiB in CSV format, SF3 weighs roughly 3 GiB and so on and so forth. It is important to note that for a given scale factor, data sets generated using different serializers contain exaclty the same data, the only difference is in how they are represented.[3] The proposed SFs are the following: 1, 3, 10, 30, 100, 300, 1000. Additionally, three small SFs, 0.003 (for quick tests of data loader components), 0.1, and 0.3 are provided to help initial validation efforts.

The Test Sponsor may select the SF that better fits their needs, by properly configuring the Datagen, as described in Section 3.3. The size of the resulting dataset is mainly affected by the following configuration parameters: the number of persons and the number of years simulated. By default, all SFs are defined over a period of three years, starting from 2010, and SFs are computed by scaling the number of Persons in the network. Table 3.12 shows some metrics of SFs 0.1, ..., 1000 data sets.

Scale Factor	0.1	0.3	1	3	10	30	100	300	1 000	3 000	10 000	30 000

Table 3.12: Properties of data sets for each scale factor in the BI workload produced by the Spark-based generator. TODO. For detailed statistics, see Table B.2

Table D.2 shows how each CSV serializer handles attributes/edges of different cardinalities, demonstrating why csv-singular-projected-fk has the most files and csv-composite-merged-fk has the least number of files.

3.4.2 Serializers

The datasets are generated in the `social_network/` directory, split into static and dynamic parts (Figure 3.1). The filenames (without the extension) end in `_i_j` where `i` is the block id and `j` is the partition id (set by `numThreads`). The SUT has to take care only of the generated Dataset to be bulk loaded. Using `NULL` values for storing optional values is allowed.

Datagen currently only supports CSV-based serializers. These produce CSV-like text files which use the pipe character "`|`" as the primary field separator and the semicolon character "`;`" as a separator for multi-valued attributes (for the composite serializers). The CSV files are stored in two subdirectories: `static/` and `dynamic/`. Depending on the number of threads used for generating the dataset, the number of files varies, since there is a file generated per thread. We indicate with "$*$" in the specification.

The following CSV variants are supported:

- csv-composite-projected-fk: Each relation with a cardinality larger than one are output in a separate file. Generated files are summarized in Table 3.14.
- csv-composite-merged-fk: This serializer is similar to csv-singular-projected-fk, but relations that have a cardinality of 1-to-N are merged in the entity files as a foreign keys. There are 13 such relations in total:
  - Comment_hasCreator_Person, Comment_isLocatedIn_Country, Comment_replyOf_Comment, Comment_replyOf_Post (merged to Comment);
  - Forum_hasModerator_Person (merged to Forum);

[2] This way of characterizing the size of the data set is identical to the scaling of TPC-H and TPC-DS.

[3] Naturally, there are slight differences in the disk usage of the data sets created with different serializers. For example, for a given scale factor, the disk usage of the data set serialized with the csv-singular-projected-fk serializer is expected to be higher, while with the csv-composite-merged-fk, it is expected to be lower.

C	File	Content
N	static/Organisation_*.csv	id | type | name | url
E	static/Organisation_isLocatedIn_Place_*.csv	OrganisationId | PlaceId
N	static/Place_*.csv	id | name | url | type
E	static/Place_isPartOf_Place_*.csv	Place1Id | Place2Id
N	static/Tag_*.csv	id | name | url
E	static/Tag_hasType_TagClass_*.csv	TagClass1Id | TagClass2Id
N	static/TagClass_*.csv	id | name | url
E	static/TagClass_isSubclassOf_TagClass_*.csv	TagId | TagClassId
N	dynamic/Comment_*.csv	creationDate | id | locationIP | browserUsed | content | length
E	dynamic/Comment_hasCreator_Person_*.csv	creationDate | CommentId | PersonId
E	dynamic/Comment_hasTag_Tag_*.csv	creationDate | CommentId | TagId
E	dynamic/Comment_isLocatedIn_Country_*.csv	creationDate | CommentId | CountryId
E	dynamic/Comment_replyOf_Comment_*.csv	creationDate | Comment1Id | Comment2Id
E	dynamic/Comment_replyOf_Post_*.csv	creationDate | CommentId | PostId
N	dynamic/Forum_*.csv	creationDate | id | title
E	dynamic/Forum_containerOf_Post_*.csv	creationDate | ForumId | PostId
E	dynamic/Forum_hasMember_Person_*.csv	creationDate | ForumId | PersonId
E	dynamic/Forum_hasModerator_Person_*.csv	creationDate | ForumId | PersonId
E	dynamic/Forum_hasTag_Tag_*.csv	creationDate | ForumId | TagId
N	dynamic/Person_*.csv	creationDate | id | firstName | lastName | gender | birthday | locationIP | browserUsed | language | email
E	dynamic/Person_hasInterest_Tag_*.csv	creationDate | PersonId | TagId
E	dynamic/Person_isLocatedIn_City_*.csv	creationDate | PersonId | CityId
E	dynamic/Person_knows_Person_*.csv	creationDate | Person1Id | Person2Id
E	dynamic/Person_likes_Comment_*.csv	creationDate | PersonId | CommentId
E	dynamic/Person_likes_Post_*.csv	creationDate | PersonId | PostId
E	dynamic/Person_studyAt_University_*.csv	creationDate | PersonId | UniversityId | classYear
E	dynamic/Person_workAt_Company_*.csv	creationDate | PersonId | CompanyId | workFrom
N	dynamic/Post_*.csv	creationDate | id | imageFile | locationIP | browserUsed | language | content | length
E	dynamic/Post_hasCreator_Person_*.csv	creationDate | PostId | PersonId
E	dynamic/Post_hasTag_Tag_*.csv	creationDate | PostId | TagId
E	dynamic/Post_isLocatedIn_Country.csv	creationDate | PostId | CountryId

Table 3.14: Files output by the `csv-composite-projected-fk` serializer (31 in total). The first part of the table contains the static entites, the second part contains the dynamic ones. Notation – C: entity category, N: node, E: edge. Warning: column names have been changed in Datagen. This document will be updated later. Until then, check the Datagen code/output for column names.

  - Organisation_isLocatedIn_Place (merged to Organisation);
  - Person_isLocatedIn_City (merged to Person);
  - Place_isPartOf_Place (merged to Place);
  - Post_hasCreator_Person, Post_isLocatedIn_Country, Forum_containerOf_Post (merged to Post);
  - Tag_hasType_TagClass (merged to Tag);
  - TagClass_isSubclassOf_TagClass (merged to TagClass)

  Generated files are summarized in Table 3.15.
- csv-singular-merged-fk: Similar to the csv-singular-projected-fk format but each entity, and relations with a cardinality larger than one, are output in a separate file. Multi-valued attributes (`Person.email` and `Person.speaks`) are stored as separate directories (`Person_email_EmailAddress` and `Person_speaks_Language`, resp.).
- csv-singular-projected-fk: Multi-valued attributes (`Person.email` and `Person.speaks`) are stored as separate directories (`Person_email_EmailAddress` and `Person_speaks_Language`, resp.).
- `raw` mode: The file names are the same as in `composite-merged-fk` but there are two important differences: (1) additional attributes, e.g. `deletionDate`, `explicitlyDeleted`, and `weight` (used for weighted graphs in Graphalytics [34]), are included, (2) all data is included, i.e. if a Forum is created and deleted before sampling the initial data set, it is included in this data set. Generated files are summarized in Table 3.16.

The inheritance hierarchies in the schema have two important characteristics (1) all subclasses use the same id space, e.g. there cannot be a Comment and a Post with id 1 at the same time, (2) they are serialized to CSVs

C	File	Content
N	static/Organisation_*.csv	id | type | name | url | LocationPlaceId
N	static/Place_*.csv	id | name | url | type | PartOfPlaceId
N	static/Tag_*.csv	id | name | url | TypeTagClassId
N	static/TagClass_*.csv	id | name | url | SubclassOfTagClassId
N	dynamic/Comment_*.csv	creationDate | id | locationIP | browserUsed | content | length | CreatorPersonId | LocationCountryId | ParentPostId | ParentCommentId
E	dynamic/Comment_hasTag_Tag_*.csv	creationDate | CommentId | TagId
N	dynamic/Forum_*.csv	creationDate | id | title | ModeratorPersonId
E	dynamic/Forum_hasMember_Person_*.csv	creationDate | ForumId | PersonId
E	dynamic/Forum_hasTag_Tag_*.csv	creationDate | ForumId | TagId
N	dynamic/Person_*.csv	creationDate | id | firstName | lastName | gender | birthday | locationIP | browserUsed | LocationCityId | language | email
E	dynamic/Person_hasInterest_Tag_*.csv	creationDate | PersonId | TagId
E	dynamic/Person_knows_Person_*.csv	creationDate | Person1Id | Person2Id
E	dynamic/Person_likes_Comment_*.csv	creationDate | PersonId | CommentId
E	dynamic/Person_likes_Post_*.csv	creationDate | PersonId | PostId
E	dynamic/Person_studyAt_University_*.csv	creationDate | PersonId | UniversityId | classYear
E	dynamic/Person_workAt_Company_*.csv	creationDate | PersonId | CompanyId | workFrom
N	dynamic/Post_*.csv	creationDate | id | imageFile | locationIP | browserUsed | language | content | length | CreatorPersonId | ContainerForumId | LocationCountryId
E	dynamic/Post_hasTag_Tag_*.csv	creationDate | PostId | TagId

Table 3.15: Files output by the `csv-composite-merged-fk` serializer (18 in total). The first part of the table contains the static entites, the second part contains the dynamic ones. Notation – C: entity category, N: node, E: edge. Warning: column names have been changed in Datagen. This document will be updated later. Until then, check the Datagen code/output for column names.

using either the *map hierarchy to single table* or *map each concrete class to its own table* strategies[4]:

Message = Comment | Post *Map each concrete class to its own table* is used i.e. separate CSV files are used for the Comment and the Post classes.

Place = City | Country | Continent *Map hierarchy to single table* is used, i.e. all Place node are serialized in a single file. A discriminator attribute "type" is used with the value denoting the concrete class, e.g. "Country".

Organisation = Company | University *Map hierarchy to single table* is used, i.e. all Organisation nodes are serialized in a single fiel. A discriminator attribute "type" is used with the value denoting the concrete class, e.g. "Company".

3.4.3 Interactive Update Streams (inserts)

The generic schema for the Interactive update streams is given in Table D.8, while the concrete schemas of each insert operations is given in Table D.9. The update stream files are generated in the `social_network/` directory and are grouped as follows:

- `updateStream_*_person.csv` files contain update operation 1: `INS 1`
- `updateStream_*_forum.csv` files contain update operations 2–8: `INS 2` `INS 3` `INS 4` `INS 5` `INS 6` `INS 7` `INS 8`

Remark: update streams in version 0.3.0 only contain inserts. Delete operations are being designed and will be released later.

3.4.4 Substitution Parameters

The substitution parameters are generated in the `substitution_parameters/` directory. Each parameter file is named `{interactive|bi}_<id>_param.txt`, corresponding to an operation of Interactive complex reads (`IC 1` –

[4] `http://www.agiledata.org/essays/mappingObjects.html`

C	File	Content
N	static/Organisation/part_*_*.csv	id | type | name | url | LocationPlaceId
N	static/Place/part_*_*.csv	id | name | url | type | PartOfPlaceId
N	static/Tag/part_*_*.csv	id | name | url | SubclassOfTagClassId
N	static/TagClass/part_*_*.csv	id | name | url | TypeTagClassId
N	dynamic/Comment/part_*_*.csv	creationDate | deletionDate | explicitlyDeleted | id | locationIP | browserUsed | content | length | CreatorPersonId | LocationCountryId | ParentPostId | ParentCommentId
E	dynamic/Comment_hasTag_Tag/part_*_*.csv	creationDate | deletionDate | CommentId | TagId
N	dynamic/Forum/part_*_*.csv	creationDate | deletionDate | explicitlyDeleted | id | title | ModeratorPersonId
E	dynamic/Forum_hasMember_Person/part_*_*.csv	creationDate | deletionDate | explicitlyDeleted | ForumId | PersonId
E	dynamic/Forum_hasTag_Tag/part_*_*.csv	creationDate | deletionDate | ForumId | TagId
N	dynamic/Person/part_*_*.csv	creationDate | deletionDate | explicitlyDeleted | id | firstName | lastName | gender | birthday | locationIP | browserUsed | LocationCityId | language | email
E	dynamic/Person_hasInterest_Tag/part_*_*.csv	creationDate | deletionDate | PersonId | TagId
E	dynamic/Person_knows_Person/part_*_*.csv	creationDate | deletionDate | explicitlyDeleted | Person1Id | Person2Id
E	dynamic/Person_likes_Comment/part_*_*.csv	creationDate | deletionDate | explicitlyDeleted | PersonId | CommentId
E	dynamic/Person_likes_Post/part_*_*.csv	creationDate | deletionDate | explicitlyDeleted | PersonId | PostId
E	dynamic/Person_studyAt_University/part_*_*.csv	creationDate | deletionDate | PersonId | UniversityId | classYear
E	dynamic/Person_workAt_Company/part_*_*.csv	creationDate | deletionDate | PersonId | CompanyId | workFrom
N	dynamic/Post/part_*_*.csv	creationDate | deletionDate | explicitlyDeleted | id | imageFile | locationIP | browserUsed | language | content | length | CreatorPersonId | ContainerForumId | LocationCountryId
E	dynamic/Post_hasTag_Tag/part_*_*.csv	creationDate | deletionDate | PostId | TagId

Table 3.16: Directories created by the raw serializer (18 in total). The first part of the table contains the static entites, the second part contains the dynamic ones. Notation – C: entity category, N: node, E: edge. The entities with the `explicitlyDeleted` attribute – Comment, Forum, Post nodes, and hasMember, knows, likes (Comment/Post) edges – denote whether the entity is deleted as part of an explicit delete operation (`DEL 2`–`DEL 8`) or implicitly through a cascading delete operation. Deletions targeting Person nodes (`DEL 1`) are always explicit, hence Person nodes have no `explicitlyDeleted` attribute. Warning: column names have been changed in Datagen. This document will be updated later. Until then, check the Datagen code/output for column names.

`IC 14`) and BI reads (`BI 1` – `BI 20`). The schemas of these files are defined by the operator, e.g. the schema of `IC 1` is "`personId|firstName`".

3.5 Introducing Delete Operations

Challenge for systems To support deletion operations graph processing systems need to solve numerous technical challenges:

1. Users should be able to *express deletion operations* using the database API, preferably using a high-level declarative query language with clear semantics [29].
2. Deletion operations *limit the algorithms and data structures* that can be used by a system. Certain dynamic graph algorithms are significantly more expensive to recompute in the presence of deletes [56] or only support either insert or deletions but not both [57]. A number of updatable matrix storage formats only support efficient insertions but not deletions [18]. Meanwhile some graph databases might be able to exploit indices to speed up deletions [15, Sec. 4.4.2]
3. *Distributed graph databases* need to employ specialized protocols to enforce consistency of deletions [67].

Challenge for benchmarks Due to their importance and challenging nature, we found it necessary to incorporate delete operations into LDBC benchmarks. However, doing so is a non-trivial task as it impacts on each component in the benchmark workflow: workload specifications, data generation, parameter curation, and the workload driver. This section focuses primarily data generation.

The initial step in generating delete operations is to define the semantics of the desired operations. To understand common behaviour of deletes we informally surveyed several social networks, the findings of which motivated the design of 8 delete operations described in Section 6.2.

The next step was to generate *delete events* within LDBC's synthetic data generator and ensure that they follow a logic order in the social network. For example, a delete knows edge event can only occur after both Persons join the network and become friends, but before either Person leaves the network. To achieve this Datagen was extended to support *dynamic entities*. Dynamic entities have a *creation date* and a *deletion date*, which together comprise an entity's *lifespan*. Once generated this allows for the extraction of deletion operations,

which can be utilized by LDBC workloads. Deriving valid lifespans for dynamic entities was the subject of a short paper published at the GRADES-NDA 2020 workshop [68] and is presented in Section 3.6.

Next it was important to distinguish between *implicit* and *explicit* delete events. Continuing with the knows edge example, once created the connection exists until either Person leaves the network, at which point the connection is implicitly deleted, as per the semantics of delete Person (Section 6.2.2). Alternatively, at any time up until this event, the friendship can be explicitly deleted, i.e. two people have a disagreement and "unfriend" each other, but both continue using the social network. Distinguishing between these types of events is important as only explicit delete events should become delete operations in the workload.

To achieve this each dynamic entity is assigned a probability of being explicitly deleted, if selected the entity is marked as such; this is used to ensure the correct serialization of delete events into delete operations. For entities selected for explicit deletion the next step is to determine a realistic time at which the event occurs. For example, a post has a higher probability of being deleted soon after it was posted compared to after 5 days. To achieve this each dynamic entity is assigned a realistic distribution to select delete event timestamps from, which respects the bounds imposed by the valid lifespans. The probability distributions used to determine if a dynamic entities is explicitly deleted and then when that event occurs is discussed in Section 3.7.

Once generated dynamic entities must be correctly serialized. Depending on its creation date, deletion date, and if the entity is explicitly deleted it can, (i) spawn an insert and delete operation, (ii) be included in the bulk load component and spawn a delete operation, (iii) just be included in the bulk load component, (iv) spawn only an insert operation, and (v) not be serialized at all! The approach for doing this is presented in Section 3.8.

We summarize the numerous challenges supporting the generation of dynamic entities and thus delete operations poses below:

1. **Validity.** The generator should produce *valid lifespans*, where each generated dynamic entity guarantees that (a) events in the graph follow a logical order: e.g. in a social network, two people can become friends only after both persons joined the network and before either person leaves the network, (b) the graph never violates the cardinality constraints prescribed by its schema, and (c) the graph continuously satisfies the semantic constraints required by the application domain (e.g. no isolated comments in a social network).
2. **Realism.** The generator should create a graph with a realistic correlations and distribution of entities over time. For example, in a social network the distribution of activity is non-uniform over time, real-world events such as elections or controversial posts can drive spikes of posts and unfollowings respectively [48]. In addition, deletions can be correlated with certain attributes: e.g. the likelihood a person leaves the network may be correlated with their number of friends [43]. Also, there are often temporal correlations between entity creation and deletion: e.g. posts have an increased chance of deletion immediately following creation compared to after a 3 month period.
3. **Serialization.** Care must be taken to distinguish between implicit and explicit delete events when creating the bulk load component, insert operations, and delete operations.
4. **Scalability.** A graph with dynamic entities should be generated at scale (up to billions of edges).

3.6 Lifespan Management

This section is based on the short paper published at the GRADES-NDA 2020 workshop [68] authored by the task force members.

In this section, we define the constraints for generating dynamic entities in a social network. Each dynamic entity gets a *lifespan*, represented by two *lifespan attributes*, a *creation date* and a *deletion date*. We first briefly review the data generator, introduce our notation and define the parameters of the generation process. Then, we define the semantic constraints which regulate the participation in certain relationships along with the constraints for selecting intervals. We illustrate an application of these with two examples, shown in Figure 3.5 and Figure 3.6.

Graph schema The LDBC Datagen component [53, 54] is responsible for generating the graph used in the benchmarks. It produces a synthetic dataset modelling a social network's activity. Its graph schema has 11 con-

crete node types connected by 20 edge types, and its entities (nodes/edges) are classified as either dynamic or static (Figure 3.1). The dynamic part of the graph comprises of a fully connected Person graph and a number of Message trees under Forums.

Notation To describe lifespans and related constraints, we use the following notation. Constants are uppercase bold, e.g. **NC**. Entity types are monospaced, e.g. Person, hasMember. Variables are lowercase italic, e.g. $pers, hm$. Entities are sans-serif, e.g. $\mathsf{P}_1, \mathsf{HM}$. For an entity x, $*x$ denotes its creation date, while $\dagger x$ denotes its deletion date. In most cases, both the creation and the deletion date are selected from an interval, e.g. $*x \in [d_1, d_2)$ means that entity x should be created between dates d_1 (inclusive) and d_2 (exclusive). The selected creation and deletion dates together form an interval that represents the lifespan of its entity. If any of the intervals for selecting the lifespan attributes of an entity are empty, i.e. $d_2 \leq d_1$, the entity should be discarded. As illustrated later, this interval is often used to determine the intervals where the creation and deletion dates of dependant entities are selected.

Parameters We parameterize the generator as follows. The network is created in 2010 and exists for 10 years at which point the network collapses (**NC** = 2020). Data is simulated for a 3-year period, between the simulation start, **SS** = 2010 and the simulation end, **SE** = 2013. In order to allow *windowed execution* by the LDBC SNB driver (used for multi-threaded and distributed operation), we define a sufficiently large amount of time that needs to pass between consecutive operations on an entity as Δ = 10s.

3.6.1 General Rules

In this section, we define general rules that must be satisfied by all entities in the graph. In subsequent sections, we refine these with domain-specific constraints. For a node n_1, we always require that:

- $*n_1 \in [\mathbf{SS}, \mathbf{SE})$, the node must be created between the simulation start and the simulation end.
- $\dagger n_1 \in [*n_1 + \Delta, \mathbf{NC})$, the node must exist for at least Δ time and must be deleted before the network collapse.

To enforce referential integrity constraints (i.e. prevent dangling edges), the lifespan of edge e between nodes n_1 and n_2 must always satisfy the following criteria:

- $*e \in [\max(*n_1, *n_2), \min(\dagger n_1, \dagger n_2, \mathbf{SE}))$, in other terms, the edge must be created no sooner than both of its endpoints but before any of its endpoints are deleted.
- $\dagger e \in [*e + \Delta, \min(\dagger n_1, \dagger n_2))$, i.e. the edge must exist for at least Δ time and deleted no later than any of its endpoints.

To further refine the constraints for edges, we distinguish between two main cases.
(1) The endpoints of edge e are existing node n_1 and node n_2 which is inserted at the same time as the edge:

- $*e = *n_2$
- $\dagger e = \min(\dagger n_1, \dagger n_2)$. In case of edges with *containment semantics* (node n_1 contains n_2 through edge e), node n_2 must always be deleted at the same time as edge e, therefore $\dagger e = \dagger n_2$ and $\dagger n_2 \leq \dagger n_1$.

(2) In other cases, the edge must be created when both of its endpoints already exist and must be deleted no later than them:

- $*e \in [\max(*n_1, *n_2) + \Delta, \min(\dagger n_1, \dagger n_2, \mathbf{SE}))$
- $\dagger e \in [*e + \Delta, \min(\dagger n_1, \dagger n_2))$

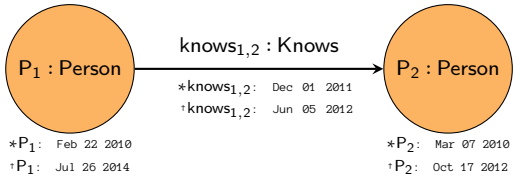
These constraints capture the "minimum" (i.e. most relaxed) set of constraints that must be enforced in all domains. Next, we introduce additional constraints specific to our social network schema.

3.6.2 Person

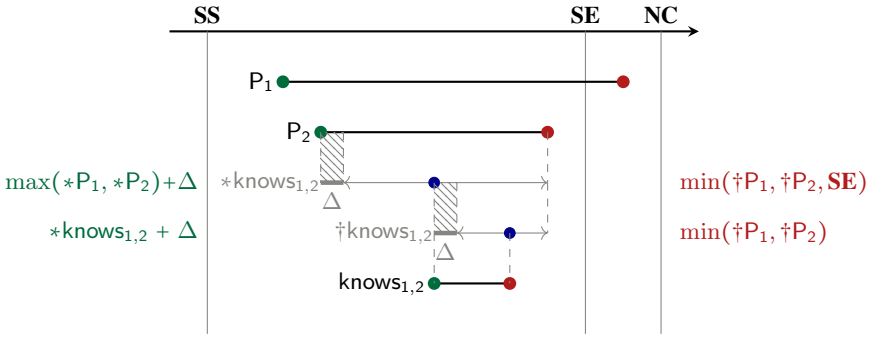
A Person p is the avatar a real-world person creates when they join the network. A Person joins the network, $*p$, during the simulation period and they leave the network, $\dagger p$, during the network lifetime:

- $*p \in \left[\mathbf{SS},\ \mathbf{SE}\right)$
- $\dagger p \in \left[*p + \Delta,\ \mathbf{NC}\right)$

For the edges of Person nodes pointing to a static node (isLocatedIn, studyAt, workAt, and hasInterest), we assign the creation and deletion date from $*p$ and $\dagger p$, resp.



(a) An instance of a knows edge connecting two Person nodes. *Creation* and *deletion* dates are shown for each entity.



(b) Illustration of the intervals in which the *creation* • and *deletion* • dates can be selected. Thick black lines represent entity lifespans and thin grey lines represent valid intervals that dates can be selected in; • indicates the selected times (spanning the lifespan interval of the given entity). On the thin grey lines, thicker sections represent the minimal amount of time that must pass before selecting a value. In case of creation dates, this is used to ensure that the dependant entity exists for at least Δ time. In case of deletion dates, it is used to ensure that the entity exists for at least Δ time.

Figure 3.5: Example graph and its intervals.

3.6.2.1 Knows

The knows edge connects two Persons p_i and p_j that know each other in the network. The intervals where the creation and deletion dates can be generated in are illustrated in Figure 3.5b and defined below:

- $*knows_{i,j} \in \left[\max(*p_i, *p_j) + \Delta,\ \min(\dagger p_i, \dagger p_j, \mathbf{SE})\right)$
- $\dagger knows_{i,j} \in \left[*knows_{i,j} + \Delta,\ \min(\dagger p_i, \dagger p_j)\right)$

3.6.3 Forum and Message

The rules for Forum and Message nodes along with their edges are given in Section 3.6.4 and Section 3.6.5, respectively, and illustrated in Figure 3.6.

3.6.4 Forum

A Forum is a meeting point where people post Messages. There exists three categories of Forums: Wall ($forum_{\mathsf{w}}$), Album ($forum_{\mathsf{a}}$), and Group ($forum_{\mathsf{g}}$). Each Forum has a set of Persons connected via hasMember edges, a set of Tags connected via hasTag edges, a single moderator connected by a hasModerator edge and a

set of Messages (discussed in Section 3.6.5). For all Forums the outgoing hasTag edges get their creation date and deletion date from $*forum$ and $\dagger forum$, respectively.

3.6.4.1 Groups

Groups are public places for people that share interests, any Person can create a Group $forum_{\mathsf{g}}$ during their lifespan. A Group can be deleted anytime after it was created.

- $*forum_{\mathsf{g}} \in \left[*p + \Delta,\ \min(\dagger p, \mathbf{SE})\right)$
- $\dagger forum_{\mathsf{g}} \in \left[*forum_{\mathsf{g}} + \Delta,\ \mathbf{NC}\right)$

Group Moderator The initial hasModerator hmd_{g} is the Group creator. If the moderator leaves the Group, the Group will have no moderator (this is allowed in the schema of version 0.4.0+, see Figure 3.1).

- $*hmd_{\mathsf{g}} \in \left[*forum_{\mathsf{g}} + \Delta,\ \min(\dagger forum_{\mathsf{g}}, \dagger p, \mathbf{SE})\right)$
- $\dagger hmd_{\mathsf{g}} \in \left[*hmd_{\mathsf{g}} + \Delta,\ \min(\dagger forum_{\mathsf{g}}, \dagger p)\right)$

Group Membership Any Person p can become a member of a given group. The hasMember hm_{g} creation is generated from the interval in which the Person and Forum lifespans overlap. The deletion date is generated from the interval between the membership creation date (incremented by Δ) and the minimum of the Person and Forum deletion dates.

- $*hm_{\mathsf{g}} \in \left[\max(*forum_{\mathsf{g}}, *p) + \Delta,\ \min(\dagger forum, \dagger p, \mathbf{SE})\right)$
- $\dagger hm_{\mathsf{g}} \in \left[*hm_{\mathsf{g}} + \Delta,\ \min(\dagger forum_{\mathsf{g}}, \dagger p)\right)$

3.6.4.2 Walls

Every Person p, has a Wall $forum_{\mathsf{w}}$ which is created when the Person joins the social network. The wall is deleted when the Person is deleted.

- $*forum_{\mathsf{w}} = *p + \Delta$
- $\dagger forum_{\mathsf{w}} = \dagger p$

Wall Moderator Each Person has a hasModerator hmd_{w} edge to their wall, which gets the creation date (incremented by Δ) and deletion date from $forum_{\mathsf{w}}$. Note, only the moderator can create Post nodes on the wall and the connecting Tag nodes are set based on the interest of the moderator.

- $*hmd_{\mathsf{w}} = *forum_{\mathsf{w}} + \Delta$
- $\dagger hmd_{\mathsf{w}} = \dagger forum_{\mathsf{w}}$

Wall Membership For a Person p_i, all their friends p_j (Person nodes connected via a knows edge) become members of $forum_{\mathsf{w}}$ at the time the knows edge is created. Hence, a hasMember hm_{w} edge gets the creation date of knows incremented by Δ. The deletion date is derived from the minimum of the Forum deletion date and knows deletion date.

- $*hm_{\mathsf{w}} = *knows_{i,j} + \Delta$
- $\dagger hm_{\mathsf{w}} = \min(\dagger forum_{\mathsf{w}}, \dagger knows_{i,j})$

3.6.4.3 Albums

A Person can create multiple Albums ($forum_{\mathsf{a}}$) containing a set of Photos. Albums can be created and then deleted at any point during the lifespan of the Person.

- $*forum_{\mathsf{a}} \in \left[*p + \Delta,\ \min(\dagger p, \mathbf{SE})\right)$
- $\dagger forum_{\mathsf{a}} \in \left[*forum_{\mathsf{a}} + \Delta,\ \dagger p\right)$

Album Moderator The Person is the moderator for any Album they create. Album ownership cannot change hence hasModerator hmd_{a} gets the creation date (incremented by Δ) and deletion date from $*forum_{\mathsf{a}}$ and $\dagger forum_{\mathsf{a}}$ respectively.

- $*hmd_{\mathsf{a}} = *forum_{\mathsf{a}} + \Delta$
- $\dagger hmd_{\mathsf{a}} = \dagger forum_{\mathsf{a}}$

Album Membership Only friends p_i of a Person p_j can become members of Albums created by p_j. The hasMember hm_{a} edge creation date is derived from the Album and knows creation dates. The deletion is derived from the Forum and knows deletion dates.

- $*hm_{\mathsf{a}} = \max(*forum_{\mathsf{a}}, *knows_{i,j}) + \Delta$
- $\dagger hm_{\mathsf{w}} = \min(\dagger forum_{\mathsf{a}}, \dagger knows_{i,j})$

3.6.5 Message

A Message is an abstract entity that represents a message created by a Person. There are two Message subtypes: Post and Comment. A Post is created in a Forum and a Comment represents a comment made by a Person to an existing Message (either a Post or a Comment). In a Forum the set of Message nodes form a *tree* with a Post node at the root and Comment nodes for the rest.

3.6.5.1 Post

A Post can be created by a Person in a Forum. Only the moderator (i.e. owner) can post on a Wall or in an Album (hasModerator), whereas all members including the moderator (hasMember/hasModerator) can post in a Group. These relationships are captured with the hm variable in the formulas. Posts are divided in three categories, *regular posts*, *photos*, and *flashmob posts*.

Regular Posts and Photos Regular posts capture the standard daily activity in a Group or on a Wall. Photos are created in Albums. (Interaction with Photos is limited to likes, see details in Section 3.6.5.3). The creation date for these is determined as follows:

$$*post \in \left[*hm + \Delta,\ \min(\dagger hm, \mathbf{SE})\right)$$

Flashmob Posts Flashmob posts are generated around events that attract significant interest (such as elections) that result in a spike in activity. These events span a 2ϕ-hour time window centered around a specific event time, flashmob event fme, in the middle of the window; there are ϕ hours each side of the specific event time.

$$*post \in \left[\max(*hm + \Delta, fme - \phi\,\mathrm{h}),\ \min(\dagger hm, fme + \phi\,\mathrm{h}, \mathbf{SE})\right)$$

The deletion dates for all categories of Posts are determined as:

$$\dagger post \in \left[*post + \Delta,\ \dagger hm\right)$$

containerOf edge Each Post node has an incoming containerOf edge which gets the same lifespan attributes as the Post.

3.6.5.2 Comment

A Comment $comm$ is created by Person p as a reply to Message m. Comments are only made in Walls and Groups. Comment always occur within γ days of their parent message following a power-law distribution with mean 6.85 hours.

- $*comm \in \left[\max(*m, *hm) + \Delta,\ \min(\dagger m, \dagger hm, *m + \gamma\,\mathrm{d}, \mathbf{SE})\right)$
- $\dagger comm \in \left[*comm + \Delta,\ \min(\dagger m, \dagger hm)\right)$

replyOf edge Comments always have an outgoing replyOf edge with containment semantics, i.e. the target Message contains the Comment. These edges get the same lifespan as their source Comment.

3.6.5.3 likes

A likes edge $likes$ can exist between Person p and Message m. Messages can only receive likes during a μ-day window after their creation at which point no more activity is generated.

- $*likes \in \left[\max(*p, *m) + \Delta,\ \min(\dagger p, \dagger m, *m + \mu\,\mathrm{d}, \mathbf{SE})\right)$
- $\dagger likes \in \left[*likes + \Delta,\ \min(\dagger p, \dagger m)\right)$

3.6.6 Complex Example

In Figure 3.6, a complex example graph is shown with the corresponding intervals. Both *the intervals for selecting the creation and deletion date* attributes and the selected *lifespan intervals* are shown.

3.7 Ensuring Realism

Capturing realistic deletion behaviour was broken down into two dimensions. Firstly, each dynamic entity is assigned a probability of being explicitly deleted. Second, if selected for explicit deletion, a deletion event date is selected using a distribution bound by the valid lifespan of that entity. To make informed choices of deletion probabilities and deletion date distributions, where possible, real-world data was used.

Delete Person Lorincz et al. [43] have analyzed iWiW, a now-defunct Hungarian social network, observing that people with more connections are less likely to leave a social network. When a Person is generated they are assigned a *maxKnows* value which indicates the amount of knows connections they will make across the lifetime of the network. This information is then utilized to determine the probability a person is explicitly deleted using the distribution provided in [43], reproduced in Figure 3.7. A deletion event date is then selected uniformly from the person's valid lifespan. On average 3.5% of Persons are deleted across the simulation period.

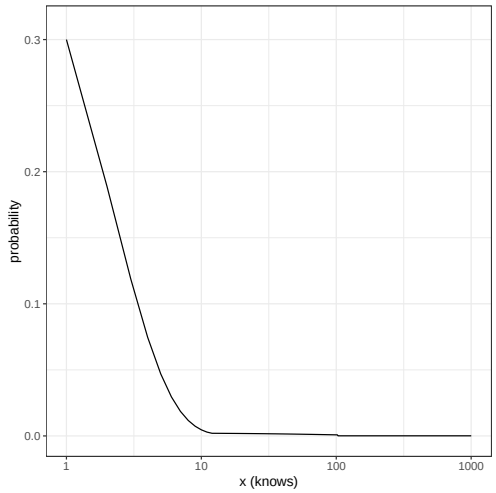
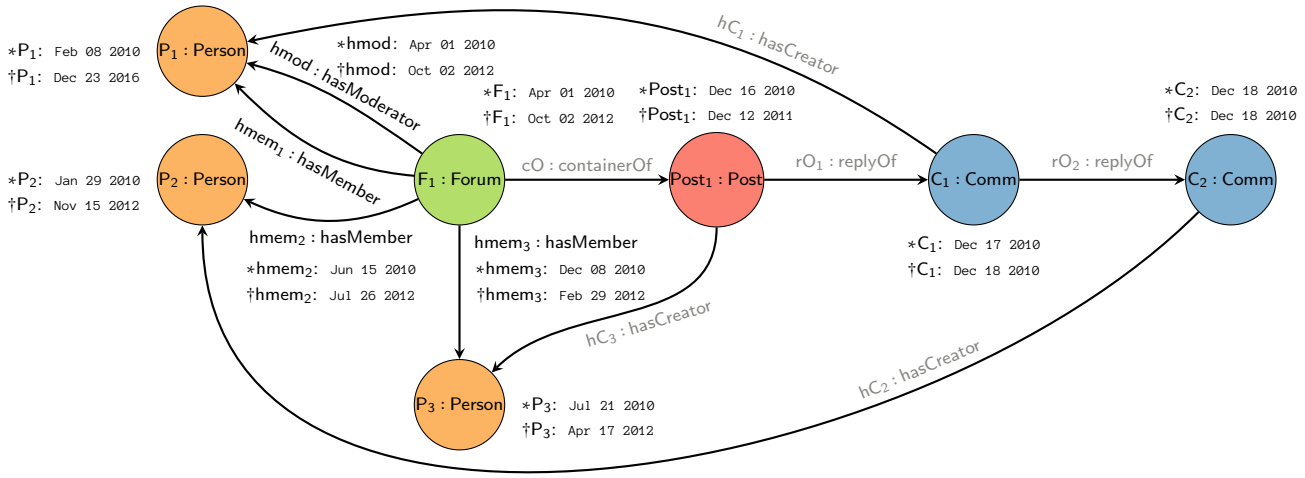
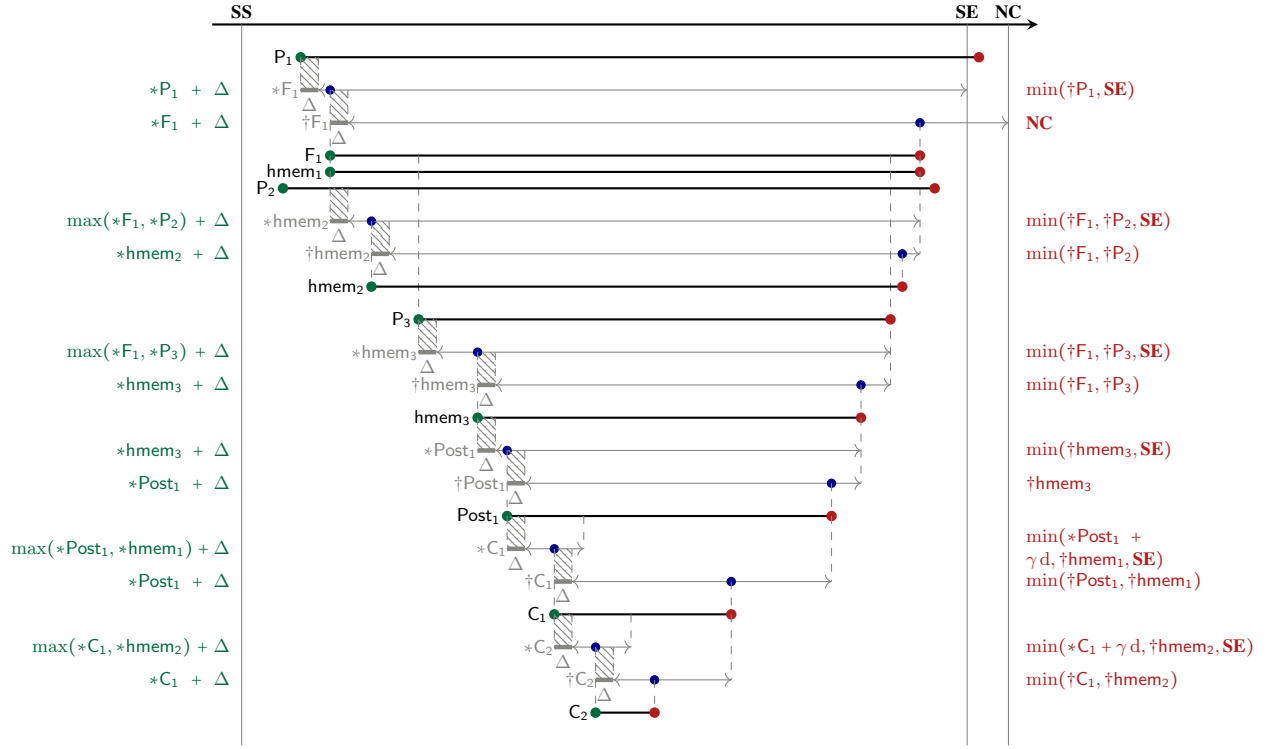


Figure 3.7: Distribution for determining the probability a Person is deleted given their number of connections.

Delete Knows Myers and Leskovec [48] analysed 1.2 billion tweets from 13.1 million Twitter users. These users made 112.3 million new connections, and deleted 39.2 million connections; a 3:1 follow:unfollow ratio. As Datagen models a generic social media platform we have chosen a different ratio of 20:1 (on average 5% of knows edges are deleted), rather than overcapture behavior that may be unique to a single site. [48] also finds a constant background flux of follows and unfollows interleaved with bursts in such activity. Currently, Datagen



(a) Example graph with an instance of a Forum containing a Message tree of depth 3 and its Person members. Lifespan attributes (*creation* and *deletion dates*) are shown for each dynamic entity. Edges in grey get their lifespan attributes as per Figure 3.1 and Section 3.6.4.



(b) Illustration of the intervals in which the *creation* • and *deletion* • dates of entities can be selected. Thick black lines represent entity lifespans and thin grey lines represent valid intervals that dates can be selected in; • indicates the selected times (spanning the lifespan interval of the given entity). On the thin grey lines, thicker sections represent the minimal amount of time that must pass before selecting a value. In case of creation dates, this is used to ensure that the dependant entity exists for at least Δ time. In case of deletion dates, it is used to ensure that the entity exists for at least Δ time.

Figure 3.6: Example graph and time intervals for selecting lifespan attributes, *creation* and *deletion dates*.

has no follow bursts, thus, we have decided not to incorporate unfollow bursts. They also find less similar friends have a high probability of being unfollowed; modelling this relationship is work in progress. If a knows edge is selected for explicit deletion then a deletion date is then selected uniformly from the edge's valid lifespan.

Delete Post/Comment and Delete Post/Comment Like Posts in groups and walls are produced via a uniform generator and a flashmob generator, capturing background events and bursts in events respectively. A comment generator is then used to produce a tree of comments on each post. Posts in albums are referred to as photos, they are produced by a different generator and do not have flashmob events nor do they have comment trees. Additionally, all posts and comments have a number of likes generated for it.

Almuhimedi et al. [2] tracked 292K Twitter users for 1 week. They found 2.4% of 67.2M tweets were deleted across 4 categories: status posts, retweets, replies, and mentions of other users that were not replies. In order to apply these findings to Datagen and obtain the average percentage of Messages and likes deleted across the simulation period, tweet categories were mapped to Datagen Message types. Table 3.17 gives the mapping and the percentage deleted across the simulation period within each category.

Message type [2]	Datagen	% Deleted
Status updates	Post/Photo	2.7
Non-reply mentions	Post/Photo	2.7
Replies	Comment	1.8
Retweets	Post/Photo/Comment Likes	2.4

Table 3.17: Mapping of [2] message types to LDBC's schema.

Additionally, [2] identified not all users delete messages, with around 50% of users doing so. Thus, each Person in the network has a 50% chance of being marked a *messageDeleter*, who subsequently, may or may not, delete post, comments, or likes. [2] also identify a relationship between the depth of replies to a tweet and the chance the tweet is deleted - a tweet with less replies is more likely to be deleted. We apply this relationship to the number of Comments in a Posts thread using the distribution in Figure 3.8. Note, this distribution has an average of 2.7% aligning with Table 3.17.

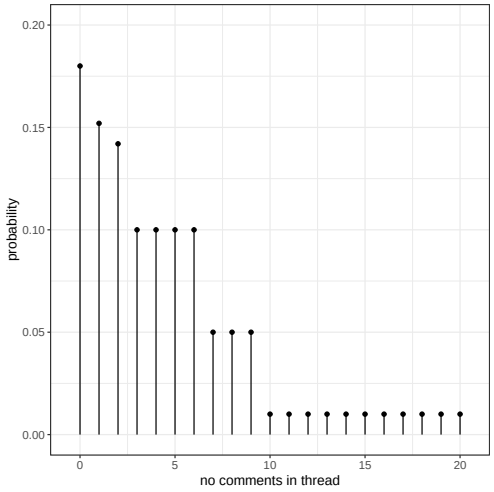


Figure 3.8: Probability a post is deleted given the number of comments in its thread.

Almuhimedi et al also observe a temporal relationship for when a tweet is deleted - a tweet has a higher chance of being deleted soon after it was created. They found 50% of all deleted tweets where removed within 8 minutes of creation. We have recreated the temporal distribution in [2] and use it to generate deletion dates from the valid lifespan intervals for posts, comments, and likes that are selected for explicit deletion Figure 3.9.

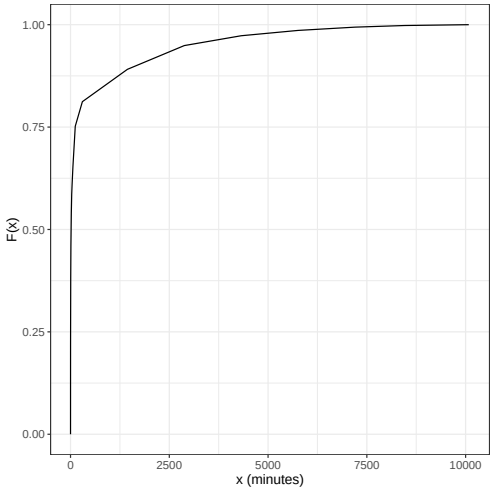


Figure 3.9: Cumulative probability density function of when a post, comment, or like is deleted after it is created (x = 0).

Delete Forum and Delete Forum Membership We currently do not have empirical evidence to motivate realistic behaviour of Forum deletion. Forums have 3 types: walls, groups, and albums. Groups and albums can be explicitly deleted, walls cannot. The target proportion of groups and albums that are deleted across the simulation period is 1%.

Additionally, we currently do not have empirical evidence to motivate realistic behaviour of hasMember edge deletion. Only membership of groups can be explicitly deleted. The target proportion of group memberships that are deleted across the simulation period is 5%.

3.8 Converting Delete Events into Delete Operations

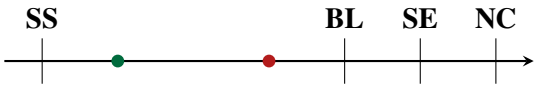
Datagen supports 3 modes, each having different output:

- **Interactive**. Produces the data necessary for the Interactive workload. Includes a set of bulk load csv files and a number of update streams, which contain only insert operations.
- **BI**. Produces the data necessary for the Business Intelligence workload. Includes a set of bulk load csv files and a number of refresh batches, which contain insert and delete operations.
- **Raw**. Produces a fully dynamic graph without insert or delete operations. Includes a set of bulk load csv files (covering whole simulation period), with each dynamic entity having creation and deletion date attributes serialized. This mode is not intended for use with any LDBC workload.

When run in Interactive mode Datagen produces a graph that monotonically increases in size over the simulation period with insert-only operations, e.g. once Person joins the network they never leave, not delete a post nor unlike a picture. This is mode is supported for backward compatibility with the Interactive workload.

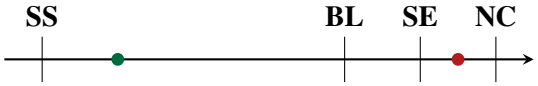
The modes BI and raw use the dynamic graph containing creation events and deletion events. Raw mode effectively serializes the graph to a bulk component and has a slightly different schema, with each entity having creation date and deletion date fields. This mode was developed for testing, yet may be useful to users that require a dynamic graph data set for purposes other than benchmarking.

For the BI mode the generated data must be converted into a bulk load component and a series of refresh batches (containing insert and delete operations). Figure 3.10 displays the possible creation and deletion dates a dynamic entity can have with respect to the bulk load cut off, simulation end, and network collapse, which determines the target file the entity should be serialized to. For example, if a Post is created after the bulk load and deleted before the simulation end this should result in a insert and a delete operation in the refresh batch data set. If an entity is marked for explicit deletion then, if the conditions in Figure 3.10 are satisfied then a deletion operation is serialized into the refresh batches.

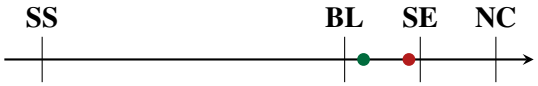


(a) Dynamic entity has creation and deletion dates before the bulk load cut off. This entity is not serialized.

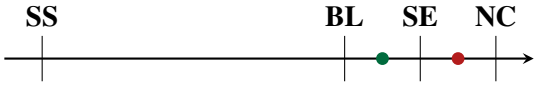
(b) Dynamic entity has creation date before the bulk load cut off and a deletion date after the bulk load cut off, but before the simulation end. Such an entity is serialized into the bulk load component and spawns a delete operation.



(c) Dynamic entity has creation date before the bulk load cut off and a deletion date after the simulation end. Such an entity is in serialized only into the bulk load component.



(d) Dynamic entity has creation date after the bulk load cut off and a deletion date before the simulation end. Such an entity produces an insert operation and a delete operation.



(e) Dynamic entity has creation date after the bulk load cut off, but before the simulation end, and a deletion date after the simulation end. Such an entity produces only an insert operation.

Figure 3.10: Possible dynamic entity *creation* • and *deletion* • dates with respect to simulation start, bulk load cut off, simulation end, and network collapse.

4 Workloads

4.1 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; *results*, a list of expected results and their types. Additionally, queries returning multiple results specify *sorting criteria* and a *limit* (to return top-k results). For strings, the sorting criteria should be interpreted as a binary comparison of the strings.[1][2]

We use the following notation:

- **Node type**: node type in the dataset. One word, possibly constructed by appending multiple words together, starting with an uppercase character and following the camel case notation, e.g. TagClass represents an entity of type "TagClass".
- **Edge type**: edge type in the dataset. One word, possibly constructed by appending multiple words together, starting with a lowercase character and following the camel case notation e.g. workAt represents an edge of type "workAt".
- **Attribute**: attribute of a node or an edge in the dataset. One word, possibly constructed by appending multiple words together, starting with a lowercase character and following the camel case notation, and prefixed by a "." to dereference the node/edge, e.g. person.firstName refers to "firstName" attribute on the "person" entity, and studyAt.classYear refers to "classYear" attribute on the "studyAt" edge.
- **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 ordered 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.

Categorization of results. Results are categorized according to their source of origin:

- **Raw** (`R`), if the result attribute is returned with an unmodified value and type.
- **Calculated** (`C`), if the result is calculated from attributes using arithmetic operators, functions, boolean conditions, etc.
- **Aggregated** (`A`), if the result is an aggregated value, e.g. a count or a sum of another value. If a result is both calculated and aggregated (e.g. `count(x) + count(y)` or `avg(x + y)`), it is considered an aggregated result.
- **Meta** (`M`), if the result is based on type information, e.g. the type of a node.

4.2 Conventions for Query Definitions

Interval notations. Closed interval boundaries are denoted with `[` and `]`, while open interval boundaries are denoted with `(` and `)`. For example, `[0, 1)` denotes an interval between 0 and 1, closed on the left and open on the right.

Comparing Date and DateTime values. Some query specifications (e.g. `BI 1`) require implementations to compare a DateTime value with a Date value. In these cases, the Date value should be implicitly converted DateTime value with a time of 00:00:00.000+00:00 (i.e. with the timezone of GMT).

[1] `C` or `POSIX` collation in PostgreSQL, see `https://www.postgresql.org/docs/13/locale.html`

[2] `BINARY` collation in DuckDB, see `https://duckdb.org/docs/sql/expressions/collations`

Matching semantics. Unless noted otherwise, the specification uses *homomorphic* matching semantics [4], i.e. both nodes and edges can occur multiple times in a match. Note that for variable-length path, duplicate edges are not allowed.

Aggregation semantics. The `count` aggregation always requires the query to determine the number of *distinct* elements (nodes or edges). For example, this can be achieved in the Cypher, SPARQL and SQL query languages with the `count(DISTINCT ...)` construct.

Graph patterns. To illustrate queries, we use graph patterns such as Figure 4.1 with the following notation:

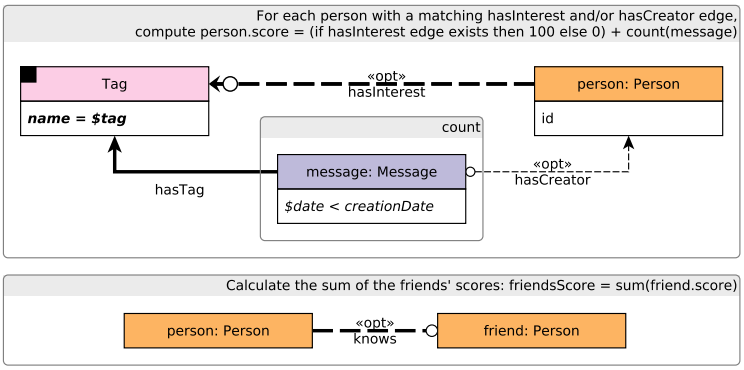


Figure 4.1: Example graph pattern.

- Nodes in the pattern are shown with rectangular boxes with their type name stated at the top and emphasized with colour coding. A black square ■ in the node's top left corner and a ***bold italic*** condition denote that the node is uniquely specified by the query parameters (e.g. by using an identifier or a unique attribute such as URL).
- Nodes in the pattern are captioned with entityName: EntityType (camel case notation for both, starting with a lowercase character for the first and an uppercase character for the second). If the entityName is neither returned in the query results (in raw, aggregated, or calculated form), nor referenced in the query specification, the entityName can be omitted.
- Edges in the graph pattern use the following notation:
  - Regular edges, i.e. edges that must be present in the subgraph, are denoted with solid black lines.
  - Negative edges, i.e. edges that must not be present in the subgraph, are denoted with dashed red lines and the «neg» keyword.
  - Optional edges, i.e. edges that may or may not be in the subgraph, are denoted with dashed black lines, the «opt» keyword, and a circle symbol ∘ at the optional end of the edge.
  - Edges without direction have no arrows. Their semantics is that there must be an edge in *the least one of the (incoming, outgoing) directions*.

- Edges with many-to-many cardinalities are denoted with thicker lines, emphasizing that they may contribute more results in the result set.
- Filtering conditions are typeset in *italic*, e.g. *id* = *$tag*.
- Attributes that should be returned are denoted in sans-serif font, e.g. name.
- Variable length paths, i.e. edges that can be traversed multiple times are denoted with *min...max, e.g. replyOf* or knows*1...2. By default, the value of min is 1, and the value of max is unlimited.
- Aggregations are shown in boxes with a grey strip on their top describing the type of aggregation (`count`, `sum`, `average`, etc.).

Keywords. The pattern notation uses a small set of keywords:

- Aggregation operations: `avg`, `count`, `sum`.
- Functions:
  - `floor(x)`: returns $\lfloor x \rfloor$,
  - `year(date)`: extracts the year from a given date,
  - `month(date)`: extracts the month from a given date.
  - `day(date)`: extracts the day (of the month) from a given date.

Deletions. Deletions of a single element are denoted with a red cross ✖, while recursive deletions are denoted with a purple cross ✖.

Resolving ambiguity. Note that if the textual description and the graph pattern are different for a particular query (either due to an error or the lack of sophistication in the graphical syntax), *the textual description takes precedence*.

4.3 Substitution Parameters

Together with the dataset, Datagen 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 selection of parameters for a query template guarantees the following properties of the resulting queries:

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

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

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

As a result, 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.

In order to get the parameter bindings for each of the queries, we have designed a *Parameter Curation* procedure that works in two stages:

1. for each query template for all possible parameter bindings, we determine the size of intermediate results in the *intended* query plan. Intermediate result size heavily influences the runtime of a query, so two queries with the same operator tree and similar intermediate result sizes at every level of this operator tree are expected to have similar runtimes. This analysis is effectively a side effect of data generation, that is we keep all the necessary counts (number of friends per user, number of posts of friends etc.) as we create the dataset.
2. then, a greedy algorithm selects ("curates") those parameters with similar intermediate result counts from the domain of all the parameters.

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. Finally, parameters for short reads are taken from those in complex reads and inserts.

4.4 Load Definition

The *Test Driver* is in charge of the execution of the Interactive Workload. At the beginning of the execution, the Test Driver creates a query mix by assigning to each query instance, a query issue time and a set of parameters taken from the generated substitution parameter set described above.

Query issue times have to be carefully assigned. Although substitution parameters are chosen in such a way that queries of the same type take similar time, not all query types have the same complexity and touch the same amount of data, which causes them to scale differently for the different scale factors. Therefore, if all query instances, regardless of their type, are issued at the same rate, those more complex queries will dominate the execution's result, making faster query types purposeless. To avoid this situation, each query type is executed at a different rate. The way the execution rate is decided, also depends on the nature of the query: complex read, short read or update.

Update queries' issue times are taken from the update streams generated by the data generator. These are the times where the actual event happened during the simulation of the social network. Complex reads' times are expressed in terms of update operations. For each complex read query type, a frequency value is assigned which specifies the relation between the number of updates performed per complex read. Table 4.1 shows the frequencies for each complex query and SF used in the Interactive workload (Chapter 5).

Query	SF1	SF3	SF10	SF30	SF100	SF300	SF1 000	SF3 000	SF10 000
1	26	26	26	26	26	26	26	TBD	TBD
2	37	37	37	37	37	37	37		
3	69	79	92	106	123	142	165		
4	36	36	36	36	36	36	36		
5	57	61	66	72	78	84	91		
6	129	172	236	316	434	580	796		
7	87	72	54	48	38	32	25		
8	45	27	15	9	5	3	1		
9	157	209	287	384	527	705	967		
10	30	32	35	37	40	44	47		
11	16	17	19	20	22	24	26		
12	44	44	44	44	44	44	44		
13	19	19	19	19	19	19	19		
14	49	49	49	49	49	49	49		

Table 4.1: Frequencies for each Interactive complex query and SF.

Finally, short reads are inserted in order to balance the ratio between reads and writes, and to simulate the behavior of a real user of the social network. For each complex read instance, a sequence of short reads is planned. There are two types of short read sequences: Person centric and Message centric. Depending on the type of the complex read, one of them is chosen. Each sequence consists of a set of short reads which are issued in a row. The issue time assigned to each short read in the sequence is determined at run time, and is based on the completion time of the complex read it depends on. The substitution parameters for short reads are taken from the results of previously executed complex reads and short reads. Once a short read sequence is issued (and provided that sufficient substitution parameters exist), there is a probability that another short read sequence is

issued. This probability decreases for each new sequence issued. Since the same random number generator seed is used across executions, the workload is deterministic.

The specified frequencies, implicitly define the query ratios between queries of different types, as well as a default target throughput. However, the Test Sponsor may specify a different target throughput to test, by "squeezing" together or "stretching" apart the queries of the workload. This is achieved by means of the "Time Compression Ratio" that is multiplied by the frequencies (see Table 4.1). Therefore, different throughputs can be tested while maintaining the relative ratios between the different query types.

5 Interactive Workload

This workload consists of a set of relatively complex read-only queries, that touch a significant amount of data – often the two-step friendship neighbourhood and associated messages –, but typically in close proximity to a single node. Hence, the query complexity is sublinear to the dataset size.

The LDBC SNB Interactive workload consists of three query classes:

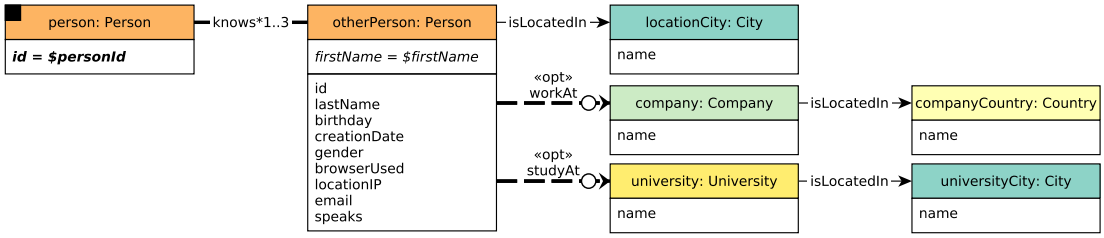
- **Complex read-only queries.** See Section 5.1.
- **Short read-only queries.** See Section 5.2.
- **Transactional update queries inserting new entities.** See Section 5.3.

A detailed description of the workload (covering reads and inserts) is available in the paper published at SIGMOD 2015 [24].

5.1 Complex Reads

Interactive / complex / 1

IC 1
IC 2
IC 3
IC 4
IC 5
IC 6
IC 7
IC 8
IC 9
IC 10
IC 11
IC 12
IC 13
IC 14

query	Interactive / complex / 1
title	Transitive friends with certain name
pattern	
desc.	Given a start Person, find Persons with a given first name (`firstName`) that the start Person is connected to (excluding start Person) by at most 3 steps via the knows relationships. Return Persons, including the distance (1..3), summaries of the Persons workplaces and places of study.

params

1	personId	ID
2	firstName	String

result

1	otherPerson.id	ID	R	
2	otherPerson.lastName	String	R	
3	distanceFromPerson	32-bit Integer	C	
4	otherPerson.birthday	Date	R	
5	otherPerson.creationDate	DateTime	R	
6	otherPerson.gender	String	R	
7	otherPerson.browserUsed	String	R	
8	otherPerson.locationIP	String	R	
9	otherPerson.email	{Long String}	R	
10	otherPerson.speaks	{String}	R	
11	locationCity.name	String	R	
12	universities	{<String, 32-bit Integer, String>}	A	{<university.name, studyAt.classYear, universityCity.name>}
13	companies	{<String, 32-bit Integer, String>}	A	{<company.name, workAt.workFrom, companyCountry.name>}

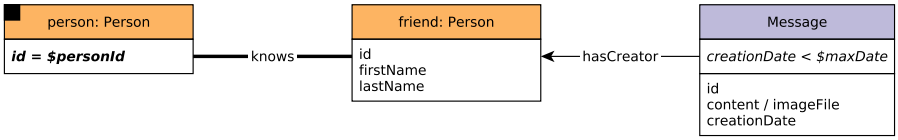
sort

1	distanceFromPerson	↑
2	otherPerson.lastName	↑
3	otherPerson.id	↑

limit	20
CPs	2.1, 5.3, 8.2
relevance	This query is a representative of a simple navigational query. It looks for paths of length 1..3 through the knows relation, starting from a given Person and ending at a Person with a given first name. It is interesting for several aspects. (1) It requires for a complex aggregation for returning the concatenation of universities, companies, languages and email information of the Person. (2) It tests the ability of the optimizer to move the evaluation of sub-queries functionally dependant on the Person, after the evaluation of the top-k. (3) Its performance is highly sensitive to properly estimating the cardinalities in each transitive path, and paying attention not to explore already visited Persons.

Interactive / complex / 2

IC 1
IC 2
IC 3
IC 4
IC 5
IC 6
IC 7
IC 8
IC 9
IC 10
IC 11
IC 12
IC 13
IC 14

query	Interactive / complex / 2
title	Recent messages by your friends
pattern	
desc.	Given a start Person (`person`), find the most recent Messages from all of that Person's friends (`friend` nodes). Only consider Messages created before the given `maxDate` (excluding that day).

params

1	`personId`	ID
2	`maxDate`	Date

result

1	`friend.id`	ID	R
2	`friend.firstName`	String	R
3	`friend.lastName`	String	R
4	`message.id`	ID	R
5	`message.content` or `message.imageFile` (for photos)	Text	R
6	`message.creationDate`	DateTime	R

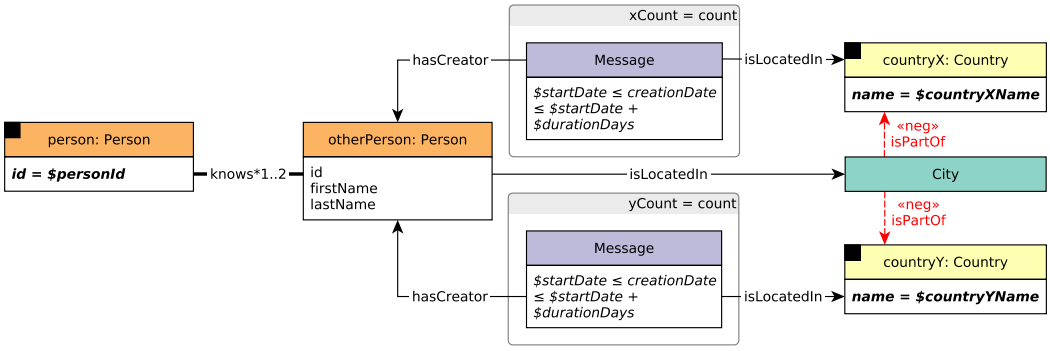
sort

1	`message.creationDate`	↓
2	`message.id`	↑

limit	20
CPs	1.1, 2.2, 2.3, 3.2, 8.5
relevance	This is a navigational query looking for paths of length two, starting from a given Person, going to their friends and from them, moving to their published Posts and Comments. This query exercices both the optimizer and how data is stored. It tests the ability to create execution plans taking advantage of the orderings induced by some operators to avoid performing expensive sorts. This query requires selecting Posts and Comments based on their creation date, which might be correlated with their identifier and therefore, having intermediate results with interesting orders. Also, messages could be stored in an order correlated with their creation date to improve data access locality. Finally, as many of the attributes required in the projection are not needed for the execution of the query, it is expected that the query optimizer will move the projection to the end.

Interactive / complex / 3

IC 1 IC 2 IC 3 IC 4 IC 5 IC 6 IC 7 IC 8 IC 9 IC 10 IC 11 IC 12 IC 13 IC 14

query	Interactive / complex / 3
title	Friends and friends of friends that have been to given countries
pattern	
desc.	Given a start Person, find Persons that are their friends and friends of friends (excluding start Person) that have made Posts / Comments in both of the given Countries, `CountryX` and `CountryY`, within a given period. Only Persons that are foreign to Countries `CountryX` and `CountryY` are considered, that is Persons whose location is neither `CountryX` nor `CountryY`.

params

1	`personId`	ID	
2	`countryXName`	String	
3	`countryYName`	String	
4	`startDate`	Date	Beginning of requested period
5	`durationDays`	32-bit Integer	Duration of requested period, in days. The interval `[startDate, startDate + durationDays)` is closed-open

result

1	`otherPerson.id`	ID	R	
2	`otherPerson.firstName`	String	R	
3	`otherPerson.lastName`	String	R	
4	`xCount`	32-bit Integer	A	Number of Messages from Country `CountryX` created by the Person within the given time
5	`yCount`	32-bit Integer	A	Number of Messages from Country `CountryY` created by the Person within the given time
6	`count`	32-bit Integer	A	`count` = `xCount` + `yCount`

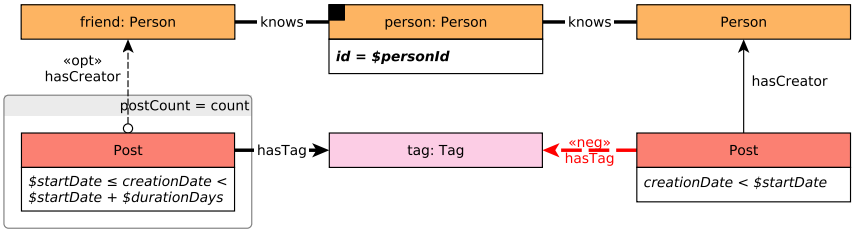
sort

1	`xCount`	↓
2	`otherPerson.id`	↑

limit	20
CPs	2.1, 3.1, 5.1, 8.2, 8.5
relevance	This query looks for paths of length two and three, starting from a Person, going to friends or friends of friends, and then moving to Messages. This query tests the ability of the query optimizer to select the most efficient join ordering, which will depend on the cardinalities of the intermediate results. Many friends of friends can be duplicate, then it is expected to eliminate duplicates and those people prior to access the Post and Comments, as well as eliminate those friends from Countries `CountryX` and `CountryY`, as the size of the intermediate results can be severely affected. A possible structural optimization could be to materialize the number of Posts and Comments created by a Person, and progressively filter those people that could not even fall in the top 20 even having all their posts in the Countries `CountryX` and `CountryY`.

Interactive / complex / 4

IC 1
IC 2
IC 3
IC 4
IC 5
IC 6
IC 7
IC 8
IC 9
IC 10
IC 11
IC 12
IC 13
IC 14

query	Interactive / complex / 4
title	New topics
pattern	
desc.	Given a start Person (`personId`), find Tags that are attached to Posts that were created by that Person's friends. Only include Tags that were attached to friends' Posts created within a given time interval, and that were never attached to friends' Posts created before this interval.

params

1	`personId`	ID	
2	`startDate`	Date	
3	`durationDays`	32-bit Integer	Duration of requested period, in days. The interval `[startDate, startDate + durationDays)` is closed-open

result

1	`tag.name`	Long String	R	
2	`postCount`	32-bit Integer	A	Number of Posts made within the given time interval that have `tag`

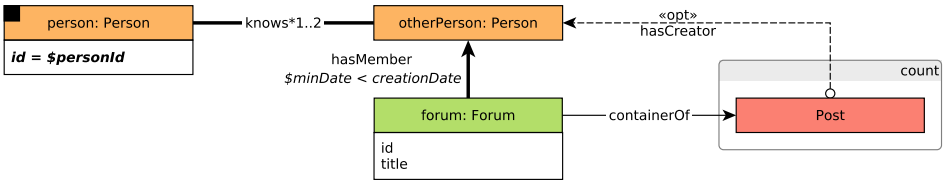
sort

1	`postCount`	↓
2	`tag.name`	↑

limit	10
CPs	2.3, 8.2, 8.5
relevance	This query looks for paths of length two, starting from a given Person, moving to Posts and then to Tags. It tests the ability of the query optimizer to properly select the usage of hash joins or index based joins, depending on the cardinality of the intermediate results. These cardinalities are clearly affected by the input Person, the number of friends, the variety of Tags, the time interval and the number of Posts.

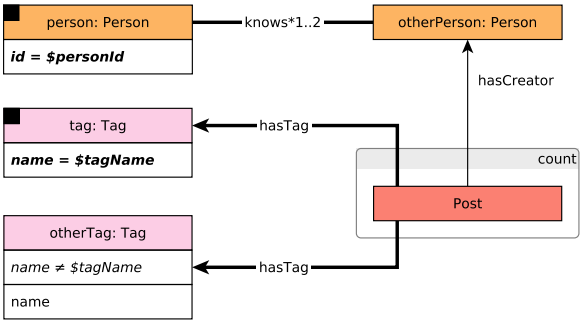
Interactive / complex / 5

IC 1
IC 2
IC 3
IC 4
IC 5
IC 6
IC 7
IC 8
IC 9
IC 10
IC 11
IC 12
IC 13
IC 14

query	Interactive / complex / 5
title	New groups
pattern	
desc.	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. For each Forum find the number of Posts that were created by any of these Persons. For each Forum and consider only those Persons which joined that particular Forum after the given date (`minDate`).
params	1 `personId` ID 2 `minDate` Date
result	1 `forum.title` Long String R 2 `postCount` 32-bit Integer A Number of Posts made in `forum` that were created by friends
sort	1 `postCount` ↓ 2 `forum.id` ↑
limit	20
CPs	2.3, 3.3, 8.2, 8.5
relevance	This query looks for paths of length two and three, starting from a given Person, moving to friends and friends of friends, and then getting the Forums they are members of. Besides testing the ability of the query optimizer to select the proper join operator, it rewards the usage of indices, but their accesses will be presumably scattered due to the two/three-hop search space of the query, leading to unpredictable and scattered index accesses. Having efficient implementations of such indices will be highly beneficial.

Interactive / complex / 6

IC 1
IC 2
IC 3
IC 4
IC 5
IC 6
IC 7
IC 8
IC 9
IC 10
IC 11
IC 12
IC 13
IC 14

query	Interactive / complex / 6
title	Tag co-occurrence
pattern	
desc.	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 both this Tag and the given Tag.

params

1	personId	ID	
2	tagName	Long String	

result

1	otherTag.name	Long String	R	
2	postCount	32-bit Integer	A	Number of Posts that were created by friends and friends of friends, which have the Tag otherTag

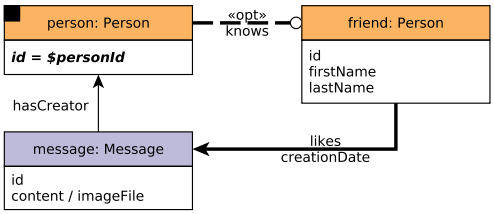
sort

1	postCount	↓
2	otherTag.name	↑

limit	10
CPs	5.1, 8.2
relevance	This query looks for paths of lengths three or four, starting from a given Person, moving to friends or friends of friends, then to Posts and finally ending at a given Tag.

Interactive / complex / 7

IC 1
IC 2
IC 3
IC 4
IC 5
IC 6
IC 7
IC 8
IC 9
IC 10
IC 11
IC 12
IC 13
IC 14

query	Interactive / complex / 7
title	Recent likers
pattern	
desc.	Given a start Person, find the most recent`likes` on any of start Person's Messages. Find Persons that liked (`likes` edge) any of start Person's Messages, the Messages they liked most recently, the creation date of that like, and the latency in minutes (`minutesLatency`) between creation of Messages and like. Additionally, for each Person found return a flag indicating (`isNew`) whether the liker is a friend of start Person. In case that a Person liked multiple Messages at the same time, return the Message with lowest identifier.
params	1 `personId` ID

result

1	`friend.id`	ID	R	
2	`friend.firstName`	String	R	
3	`friend.lastName`	String	R	
4	`likes.creationDate`	DateTime	R	
5	`message.id`	ID	R	
6	`message.content` or `message.imageFile` (for photos)	Text	R	
7	`minutesLatency`	32-bit Integer	C	Duration between creation of the Message and the creation of the like, in minutes
8	`isNew`	Boolean	C	False if `person` and `friend` know each other, True otherwise

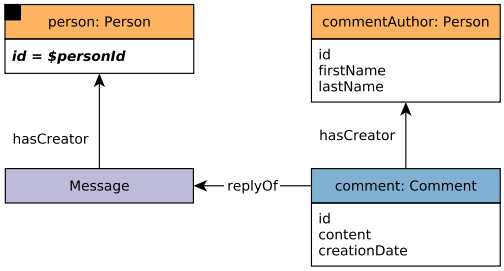
sort

1	`likes.creationDate`	↓
2	`friend.id`	↑

limit	20
CPs	2.2, 2.3, 3.3, 5.1, 8.1, 8.3
relevance	This query looks for paths of length two, starting from a given Person, moving to its published messages and then to Persons who liked them. It tests several aspects related to join optimization, both at query optimization plan level and execution engine level. On the one hand, many of the columns needed for the projection are only needed in the last stages of the query, so the optimizer is expected to delay the projection until the end. This query implies accessing two-hop data, and as a consequence, index accesses are expected to be scattered. We expect to observe variate cardinalities, depending on the characteristics of the input parameter, so properly selecting the join operators will be crucial. This query has a lot of correlated sub-queries, so it is testing the ability to flatten the query execution plans.

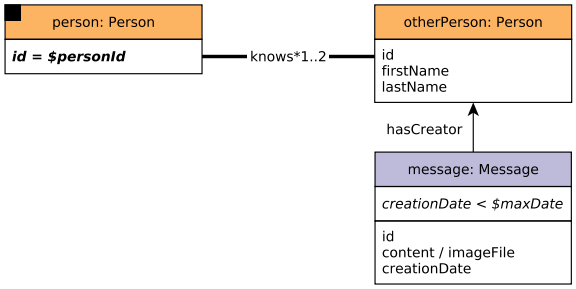
Interactive / complex / 8

IC 1
IC 2
IC 3
IC 4
IC 5
IC 6
IC 7
IC 8
IC 9
IC 10
IC 11
IC 12
IC 13
IC 14

query	Interactive / complex / 8
title	Recent replies
pattern	
desc.	Given a start Person, find the most recent Comments that are replies to Messages of the start Person. Only consider direct (single-hop) replies, not the transitive (multi-hop) ones. Return the reply Comments, and the Person that created each reply Comment.
params	1 personId ID
result	1 commentAuthor.id ID R 2 commentAuthor.firstName String R 3 commentAuthor.lastName String R 4 comment.creationDate DateTime R 5 comment.id ID R 6 comment.content Text R
sort	1 comment.creationDate ↓ 2 comment.id ↑
limit	20
CPs	2.4, 3.3, 5.3
relevance	This query looks for paths of length two, starting from a given Person, going through its created Messages and finishing at their replies. In this query there is temporal locality between the replies being accessed. Thus the top-k order by this can interact with the selection, i.e. do not consider older Posts than the 20th oldest seen so far.

Interactive / complex / 9

IC 1
IC 2
IC 3
IC 4
IC 5
IC 6
IC 7
IC 8
IC 9
IC 10
IC 11
IC 12
IC 13
IC 14

query	Interactive / complex / 9
title	Recent messages by friends or friends of friends
pattern	
desc.	Given a start Person, find the most recent Messages created by that Person's friends or friends of friends (excluding start Person). Only consider Messages created before the given `maxDate` (excluding that day).

params

1	personId	ID
2	maxDate	Date

result

1	otherPerson.id	ID	R	
2	otherPerson.firstName	String	R	
3	otherPerson.lastName	String	R	
4	message.id	ID	R	
5	message.content or message.imageFile (for photos)	Text	R	
6	message.creationDate	DateTime	R	

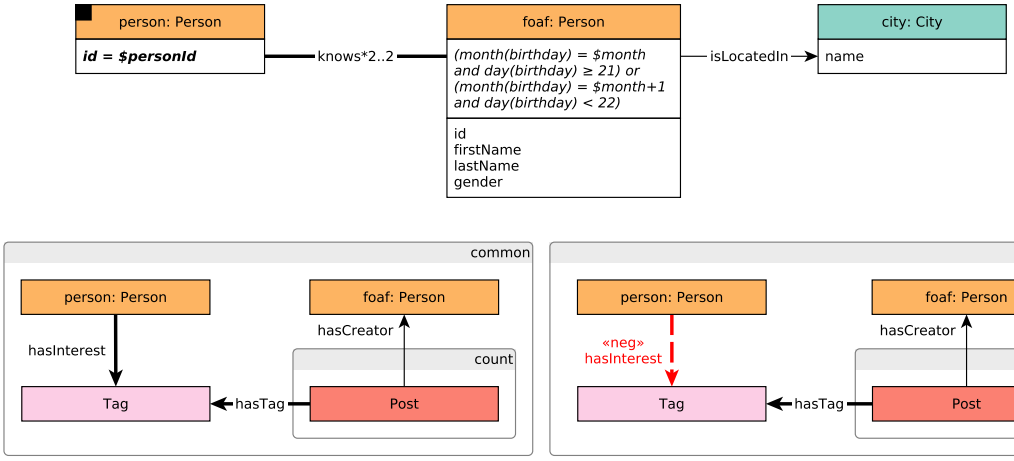
sort

1	message.creationDate	↓
2	message.id	↑

limit	20
CPs	1.1, 1.2, 2.2, 2.3, 3.2, 3.3, 8.5
relevance	This query looks for paths of length two or three, starting from a given Person, moving to its friends and friends of friends, and ending at their created Messages. This is one of the most complex queries, as the list of choke points indicates. This query is expected to touch variable amounts of data with entities of different characteristics, and therefore, properly estimating cardinalities and selecting the proper operators will be crucial.

Interactive / complex / 10

IC 1
IC 2
IC 3
IC 4
IC 5
IC 6
IC 7
IC 8
IC 9
IC 10
IC 11
IC 12
IC 13
IC 14

query	Interactive / complex / 10
title	Friend recommendation
pattern	
desc.	Given a start Person with id `personId`, find that Person's friends of friends (`foaf`) – excluding the start Person and his/her 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 `friend` and the start `person`, where `commonInterestScore` is defined as follows: • `common` = number of Posts created by `friend`, such that the Post has a Tag that the start `person` is interested in • `uncommon` = number of Posts created by `friend`, such that the Post has no Tag that the start `person` is interested in • `commonInterestScore` = `common` - `uncommon`

params

1	personId	ID	
2	month	32-bit Integer	Between 1 and 12. Implementations may also pass the next month as an additional `nextMonth` parameter

result

1	foaf.id	ID	R	
2	foaf.firstName	String	R	
3	foaf.lastName	String	R	
4	commonInterestScore	32-bit Integer	A	
5	foaf.gender	String	R	
6	city.name	String	R	

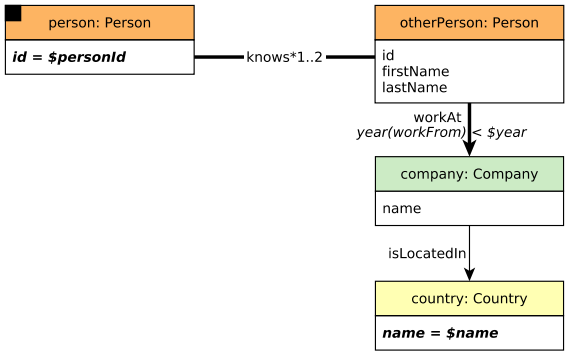
sort

1	commonInterestScore	↓	
2	foaf.id	↑	

limit	10
CPs	2.3, 3.3, 4.1, 4.2, 5.1, 5.2, 6.1, 7.1, 8.6
relevance	This query looks for paths of length two, starting from a Person and ending at the friends of their friends. It does widely scattered graph traversal, and one expects no locality of in friends of friends, as these have been acquired over a long time and have widely scattered identifiers. The join order is simple but one must see that the anti-join for "not in my friends" is better with hash. Also the last pattern in the scalar sub-queries joining or anti-joining the Tags of the candidate's Posts to interests of self should be by hash.

Interactive / complex / 11

IC 1
IC 2
IC 3
IC 4
IC 5
IC 6
IC 7
IC 8
IC 9
IC 10
IC 11
IC 12
IC 13
IC 14

query	Interactive / complex / 11
title	Job referral
pattern	
desc.	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).

params

1	personId	ID	
2	countryName	String	
3	workFromYear	32-bit Integer	

result

1	otherPerson.id	ID	R	
2	otherPerson.firstName	String	R	
3	otherPerson.lastName	String	R	
4	company.name	String	R	
5	workAt.workFrom	32-bit Integer	R	

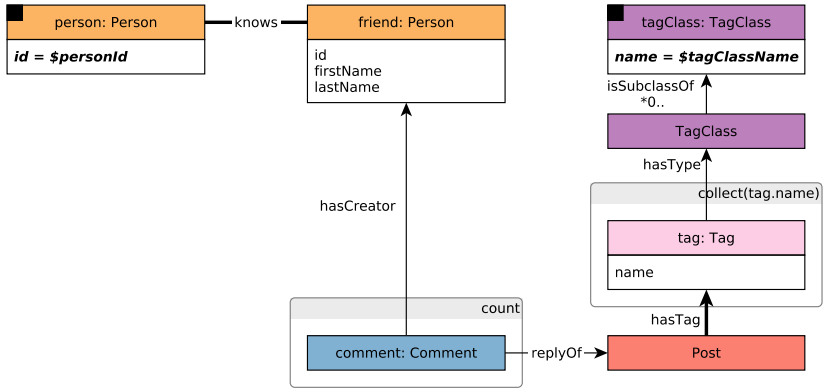
sort

1	workAt.workFrom	↑	
2	otherPerson.id	↑	
3	company.name	↓	

limit	10
CPs	1.3, 2.3, 2.4, 3.3, 4.2
relevance	This query looks for paths of length two or three, starting from a Person, moving to friends or friends of friends, and ending at a Company. In this query, there are selective joins and a top-k order by that can be exploited for optimizations.

Interactive / complex / 12

IC 1
IC 2
IC 3
IC 4
IC 5
IC 6
IC 7
IC 8
IC 9
IC 10
IC 11
IC 12
IC 13
IC 14

query	Interactive / complex / 12
title	Expert search
pattern	
desc.	Given a start Person, find the Comments that this Person's friends made in reply to Posts, considering only those Comments that are direct (single-hop) replies to Posts, not the transitive (multi-hop) ones. 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, but only collect Tags with the given TagClass or with a descendant of that TagClass. Return Persons with at least one reply, the reply count, and the collection of Tags.

params

1	personId	ID	
2	tagClassName	Long String	

result

1	friend.id	ID	R	
2	friend.firstName	String	R	
3	friend.lastName	String	R	
4	tagNames	{Long String}	A	
5	replyCount	32-bit Integer	A	

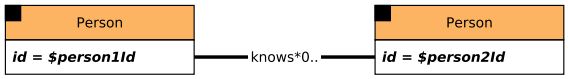
sort

1	replyCount	↓	
2	friend.id	↑	

limit	20
CPs	3.3, 7.2, 7.3, 8.2
relevance	This query starts at a Person, moves to its friends, and the to their Comments and their root Posts. Then, it gets the Tag of each Post and checks whether it (directly or transitively) belongs to the specified TagClass. This can be thought of a bidirectional search between the Person and the TagClass. The difficulty of this query is determining the optimal direction of this traversal.

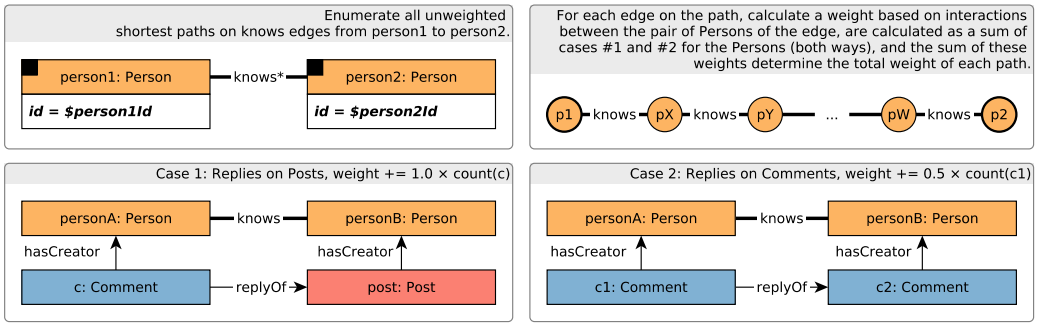
Interactive / complex / 13

IC 1
IC 2
IC 3
IC 4
IC 5
IC 6
IC 7
IC 8
IC 9
IC 10
IC 11
IC 12
IC 13
IC 14

query	Interactive / complex / 13
title	Single shortest path
pattern	
desc.	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: • −1: no path found • 0: start person = end person • > 0: path found (start person ≠ end person)
params	1 person1Id ID 2 person2Id ID
result	1 shortestPathLength 32-bit Integer C
CPs	3.3, 7.2, 7.3, 7.5, 8.1, 8.6
relevance	This query looks for a variable length path, starting at a given Person and finishing at an another given Person. Proper cardinality estimation and search space pruning, will be crucial. This query also allows for possible parallel implementations.

Interactive / complex / 14

IC 1
IC 2
IC 3
IC 4
IC 5
IC 6
IC 7
IC 8
IC 9
IC 10
IC 11
IC 12
IC 13
IC 14

query	Interactive / complex / 14
title	Trusted connection paths
pattern	
desc.	Given two Persons, find all (unweighted) shortest paths between these two Persons, in the subgraph induced by the knows relationship. Then, for each path calculate a weight. The nodes in the path are Persons, and the weight of a path is the sum of weights between every pair of consecutive Person nodes in the path. The weight for a pair of Persons is calculated based on their interactions: • Every direct reply (by one of the Persons) to a Post (by the other Person) contributes 1.0. • Every direct reply (by one of the Persons) to a Comment (by the other Person) contributes 0.5. Note that interactions are counted both ways (e.g. if Alice writes 2 Post replies and 1 Comment reply to Bob, while Bob writes 3 Post replies and 4 Comment replies to Alice, their interaction score is $2 \times 1.0 + 1 \times 0.5 + 3 \times 1.0 + 4 \times 0.5 = 7.5$). Return all the paths with shortest length, and their weights. Do not return any rows if there is no path between the two Persons.

params

1	person1Id	ID
2	person2Id	ID

result

1	personIdsInPath	[ID]	C	identifiers representing an ordered sequence of the Persons in the path
2	pathWeight	64-bit Float	C	

sort

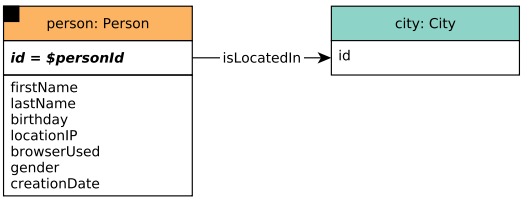
1	pathWeight	↓	The order of paths with the same weight is unspecified

CPs	3.3, 5.3, 7.2, 7.3, 7.5, 7.7, 8.1, 8.2, 8.3, 8.6
relevance	This query looks for a variable length path, starting at a given Person and finishing at an another given Person. This is a more complex query as it not only requires computing the path length, but returning it and computing a weight. To compute this weight one must look for smaller sub-queries with paths of length three, formed by the two Persons at each step, a Post and a Comment.

5.2 Short Reads

Interactive / short / 1

IS 1
IS 2
IS 3
IS 4
IS 5
IS 6
IS 7

query	Interactive / short / 1
title	Profile of a person
pattern	
desc.	Given a start Person, retrieve their first name, last name, birthday, IP address, browser, and city of residence.

params

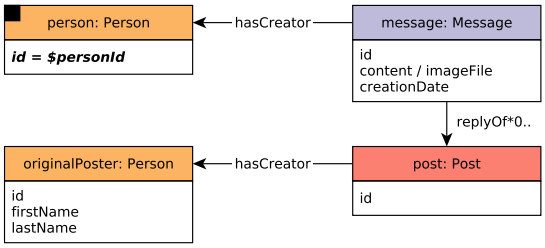
1	personId	ID	

result

1	person.firstName	String	R	
2	person.lastName	String	R	
3	person.birthday	Date	R	
4	person.locationIP	String	R	
5	person.browserUsed	String	R	
6	city.id	ID	R	
7	person.gender	String	R	
8	person.creationDate	DateTime	R	

Interactive / short / 2

IS 1
IS 2
IS 3
IS 4
IS 5
IS 6
IS 7

query	Interactive / short / 2
title	Recent messages of a person
pattern	
desc.	Given a start Person, retrieve the last 10 Messages created by that user. For each Message, return that Message, the original Post in its conversation (`post`), and the author of that Post (`originalPoster`). If any of the Messages is a Post, then the original Post (`post`) will be the same Message, i.e. that Message will appear twice in that result.

params

1	personId	ID	

result

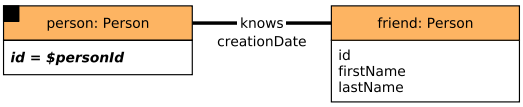
1	message.id	ID	R	
2	message.content or message.imageFile (for photos)	Text	R	
3	message.creationDate	DateTime	R	
4	post.id	ID	R	
5	originalPoster.id	ID	R	
6	originalPoster.firstName	String	R	
7	originalPoster.lastName	String	R	

sort

1	message.creationDate	↓	
2	message.id	↓	

Interactive / short / 3

IS 1
IS 2
IS 3
IS 4
IS 5
IS 6
IS 7

query	Interactive / short / 3
title	Friends of a person
pattern	
desc.	Given a start Person, retrieve all of their friends, and the date at which they became friends.

params

1	personId	ID	

result

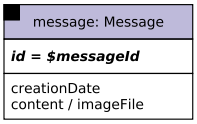
1	friend.id	ID	R	
2	friend.firstName	String	R	
3	friend.lastName	String	R	
4	knows.creationDate	DateTime	R	

sort

1	knows.creationDate	↓	
2	friend.id	↑	

Interactive / short / 4

IS 1
IS 2
IS 3
IS 4
IS 5
IS 6
IS 7

query	Interactive / short / 4
title	Content of a message
pattern	
desc.	Given a Message, retrieve its content and creation date.

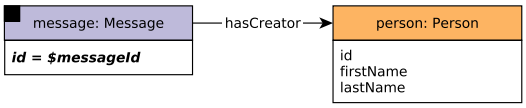
params

1	messageId	ID	

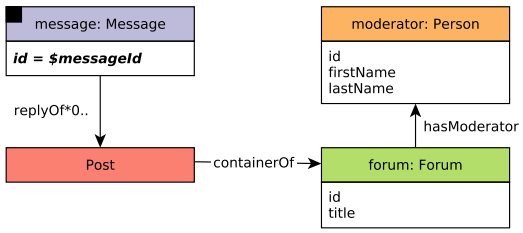
result

1	message.creationDate	DateTime	R	messageCreationDate
2	message.content or message.imageFile (for photos)	Text	R	messageContent

Interactive / short / 5

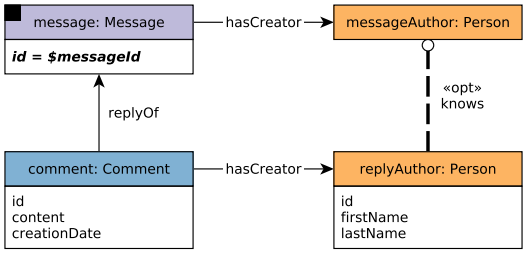
IS 1	query	Interactive / short / 5
IS 2	title	Creator of a message
IS 3	pattern	
IS 4	desc.	Given a Message, retrieve its author.
IS 5	params	1 messageId ID
IS 6	result	1 person.id ID R 2 person.firstName String R 3 person.lastName String R
IS 7		

Interactive / short / 6

IS 1	query	Interactive / short / 6
IS 2	title	Forum of a message
IS 3	pattern	
IS 4	desc.	Given a Message, retrieve the Forum that contains it and the Person that moderates that Forum. Since Comments are not directly contained in Forums, for Comments, return the Forum containing the original Post in the thread which the Comment is replying to.
IS 5	params	1 messageId ID
IS 6	result	1 forum.id ID R 2 forum.title Long String R 3 moderator.id ID R 4 moderator.firstName String R 5 moderator.lastName String R
IS 7		

Interactive / short / 7

IS 1
IS 2
IS 3
IS 4
IS 5
IS 6
IS 7

query	Interactive / short / 7
title	Replies of a message
pattern	
desc.	Given a Message, retrieve the (1-hop) Comments that reply to it. In addition, return a boolean flag knows indicating if the author of the reply (replyAuthor) knows the author of the original message (messageAuthor). If author is same as original author, return False for knows flag.

params

1	messageId	ID

result

1	comment.id	ID	R	
2	comment.content	Text	R	
3	comment.creationDate	DateTime	R	
4	replyAuthor.id	ID	R	
5	replyAuthor.firstName	String	R	
6	replyAuthor.lastName	String	R	
7	knows	Boolean	C	True if the knows edge exists between the replyAuthor and the messageAuthor nodes, False otherwise (including the case when the two nodes are the same)

sort

1	comment.creationDate	↓
2	replyAuthor.id	↑

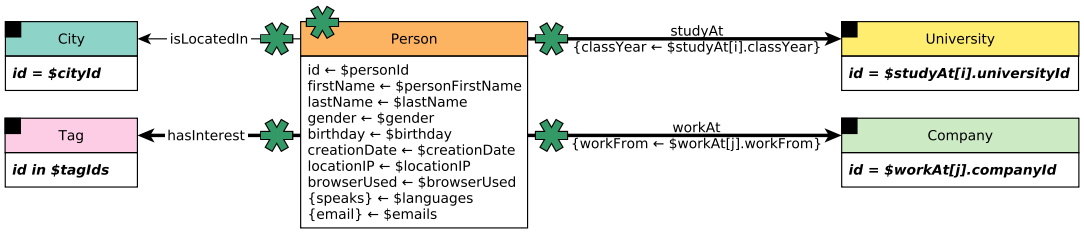
5.3 Inserts (Formerly: Updates)

Each insert query inserts

1. either a single node of a certain type, along with its edges to other existing nodes
2. or a single edge of a certain type between two existing nodes.

In versions 0.3.x, these operations were called “Interactive updates”. From 0.4.0 onwards, these are called “Interactive inserts”.

Interactive / insert / 1

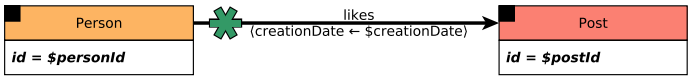
query	Interactive / insert / 1
title	Add person
pattern	
desc.	Add a Person *node*, connected to the network by 4 possible *edge* types.

params

1	personId	ID	
2	personFirstName	String	
3	personLastName	String	
4	gender	String	
5	birthday	Date	
6	creationDate	DateTime	
7	locationIP	String	
8	browserUsed	String	
9	cityId	ID	
10	languages	{String}	
11	emails	{Long String}	
12	tagIds	{ID}	
13	studyAt	{<ID, 32-bit Integer>}	{‹universityId, classYear›}
14	workAt	{<ID, 32-bit Integer>}	{‹companyId, workFrom›}

CPs	9.1, 9.2

Interactive / insert / 2

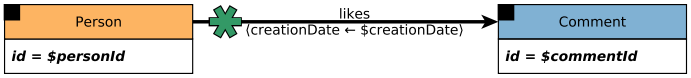
query	Interactive / insert / 2
title	Add like to post
pattern	
desc.	Add a likes *edge* to a Post.

params

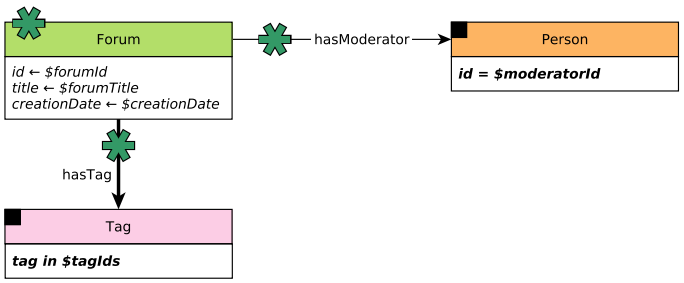
1	personId	ID	
2	postId	ID	
3	creationDate	DateTime	

CPs	9.2

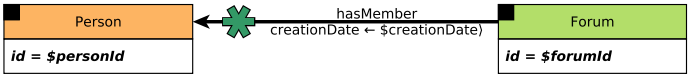
Interactive / insert / 3

query	Interactive / insert / 3
title	Add like to comment
pattern	
desc.	Add a likes *edge* to a Comment.
params	1 personId ID 2 commentId ID 3 creationDate DateTime
CPs	9.2

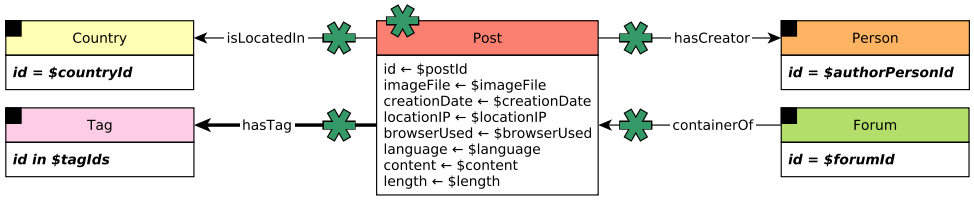
Interactive / insert / 4

query	Interactive / insert / 4
title	Add forum
pattern	
desc.	Add a Forum *node*, connected to the network by 2 possible *edge* types.
params	1 forumId ID 2 forumTitle Long String 3 creationDate DateTime 4 moderatorId ID 5 tagIds {ID}
CPs	9.1, 9.2

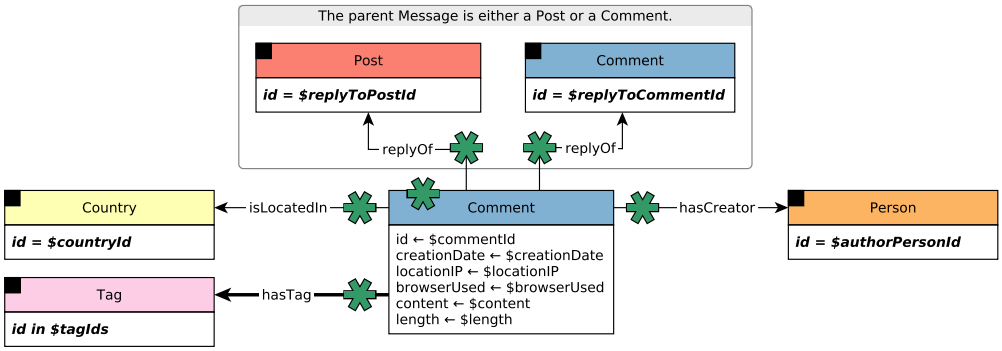
Interactive / insert / 5

query	Interactive / insert / 5	
title	Add forum membership	
pattern		
desc.	Add a Forum membership *edge* (hasMember) to a Person.	
params	1	personId	ID
	2	forumId	ID
	3	creationDate	DateTime
CPs	9.1, 9.2	

Interactive / insert / 6

query	Interactive / insert / 6	
title	Add post	
pattern		
desc.	Add a Post *node* to the social network connected by 4 possible *edge* types (hasCreator, containerOf, isLocatedIn, hasTag).	
params	1	postId	ID
	2	imageFile	String
	3	creationDate	DateTime
	4	locationIP	String
	5	browserUsed	String
	6	language	String
	7	content	Text
	8	length	32-bit Integer
	9	authorPersonId	ID
	10	forumId	ID
	11	countryId	ID
	12	tagIds	{ID}
CPs	9.1, 9.2	

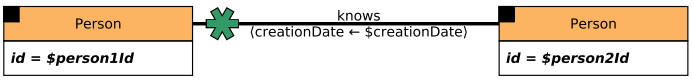
Interactive / insert / 7

query	Interactive / insert / 7
title	Add comment
pattern	
desc.	Add a Comment *node* replying to a Post/Comment, connected to the network by 4 possible *edge* types (replyOf, hasCreator, isLocatedIn, hasTag).
params	

1	commentId	ID	
2	creationDate	DateTime	
3	locationIP	String	
4	browserUsed	String	
5	content	Text	
6	length	32-bit Integer	
7	authorPersonId	ID	
8	countryId	ID	
9	replyToPostId	ID	−1 if the Comment is a reply of a Comment
10	replyToCommentId	ID	−1 if the Comment is a reply of a Post
11	tagIds	{ID}	

CPs	9.1, 9.2

Interactive / insert / 8

query	Interactive / insert / 8
title	Add friendship
pattern	
desc.	Add a friendship *edge* (knows) between two Persons.
params	

1	person1Id	ID	
2	person2Id	ID	
3	creationDate	DateTime	

CPs	9.2

6 Business Intelligence Workload

The draft BI workload was published at the GRADES-NDA workshop at SIGMOD 2018 [62]. Version 1.0 of the workload is currently being prepared.

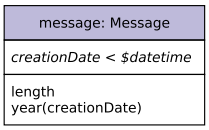
The LDBC SNB BI workload consists of two sets of operations:

- **Read queries.** Complex read queries touching a significant portion of the data. See Section 6.1.
- **Microbatches of refresh operations.** A set of insert and delete operations, batched for a given time period (e.g. an hour, a day, etc.). See Section 6.2.

6.1 Reads

BI / read / 1

BI 1 BI 2 BI 3 BI 4 BI 5 BI 6 BI 7 BI 8 BI 9 BI 10 BI 11 BI 12 BI 13 BI 14 BI 15 BI 16 BI 17 BI 18 BI 19 BI 20

query	BI / read / 1
title	Posting summary
pattern	
desc.	Given a datetime, find all Messages created before that moment. Group them by a 3-level grouping: 1. by year of creation 2. for each year, group into Message types: is Comment or not 3. for each year-type group, split into four groups based on length of their content • `0`: 0 ≤ length < 40 (short) • `1`: 40 ≤ length < 80 (one liner) • `2`: 80 ≤ length < 160 (tweet) • `3`: 160 ≤ length (long)

params

1	`datetime`	DateTime	For later microbatches, later `datetime` parameters are selected keep the variance low (<0.5%)

result

1	`year`	32-bit Integer	R	`year(message.creationDate)`
2	`isComment`	Boolean	M	True for Comments, False for Posts
3	`lengthCategory`	32-bit Integer	C	`0` for short, `1` for one-liner, `2` for tweet, `3` for long
4	`messageCount`	32-bit Integer	A	Total number of Messages in that group
5	`averageMessageLength`	32-bit Float	A	Average length of the Message content in that group
6	`sumMessageLength`	32-bit Integer	A	Sum of all Message content lengths
7	`percentageOfMessages`	32-bit Float	A	Number of Messages in group as a percentage of all messages created before the given date

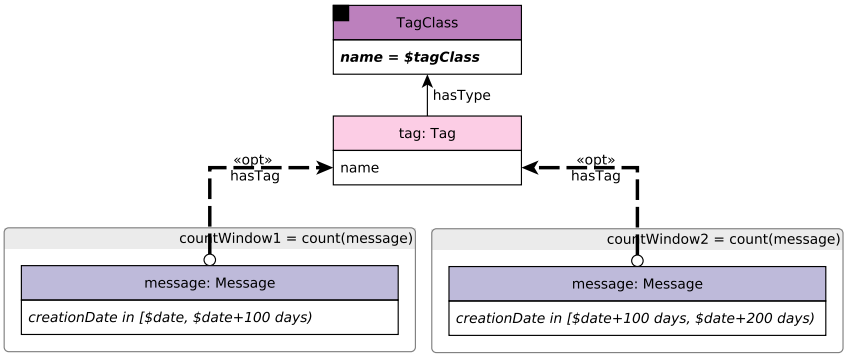
sort

1	`year`	↓	
2	`isComment`	↑	False < True, i.e. Posts come first and Comments second
3	`lengthCategory`	↑	order based on the `lengthCategory` value

limit	n/a
CPs	1.2, 3.2, 4.1, 4.2, 8.5

BI / read / 2

BI 1 BI 2 BI 3 BI 4 BI 5 BI 6 BI 7 BI 8 BI 9 BI 10 BI 11 BI 12 BI 13 BI 14 BI 15 BI 16 BI 17 BI 18 BI 19 BI 20

query	BI / read / 2
title	Tag evolution
pattern	
desc.	Find the Tags under a given TagClass that were used in Messages during in the 100-day period starting at `date` and compare it with the 100-day period that follows. For the Tags and for both months, compute the count of Messages.

params

1	`date`	Date	
2	`tagClass`	Long String	TagClasses with a similar amount of Messages are selected

result

1	`tag.name`	Long String	R	
2	`countWindow1`	32-bit Integer	A	Occurrences of the `tagClass` during the first time window
3	`countWindow2`	32-bit Integer	A	Occurrences of the `tagClass` during the second time window
4	`diff`	32-bit Integer	A	Absolute difference of `countWindow1` and `countWindow2`

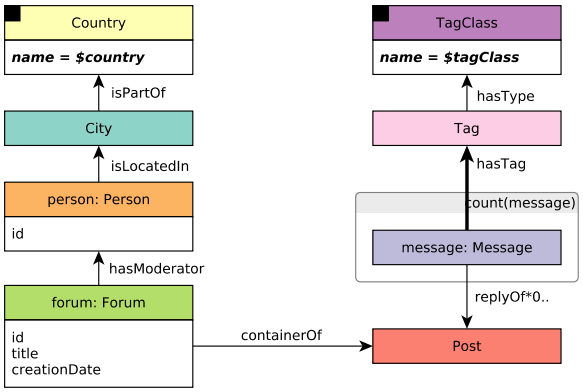
sort

1	`diff`	↓
2	`tag.name`	↑

limit	100
CPs	2.4, 3.1, 3.2, 4.1, 4.2, 4.3, 5.3, 6.1, 8.2, 8.5

BI / read / 3

BI 1
BI 2
BI 3
BI 4
BI 5
BI 6
BI 7
BI 8
BI 9
BI 10
BI 11
BI 12
BI 13
BI 14
BI 15
BI 16
BI 17
BI 18
BI 19
BI 20

query	BI / read / 3
title	Popular topics in a country
pattern	
desc.	Given a TagClass and a Country, find all the Forums created in the given Country, containing at least one Message with Tags belonging directly to the given TagClass, and count the Messages by the Forum which contains them. The location of a Forum is identified by the location of the Forum's moderator.

params

1	tagClass	Long String	TagClasses with a similar amount of Messages are selected
2	country	Long String	Big Countries are selected

result

1	forum.id	ID	R	
2	forum.title	Long String	R	
3	forum.creationDate	DateTime	R	
4	person.id	ID	R	
5	messageCount	32-bit Integer	A	

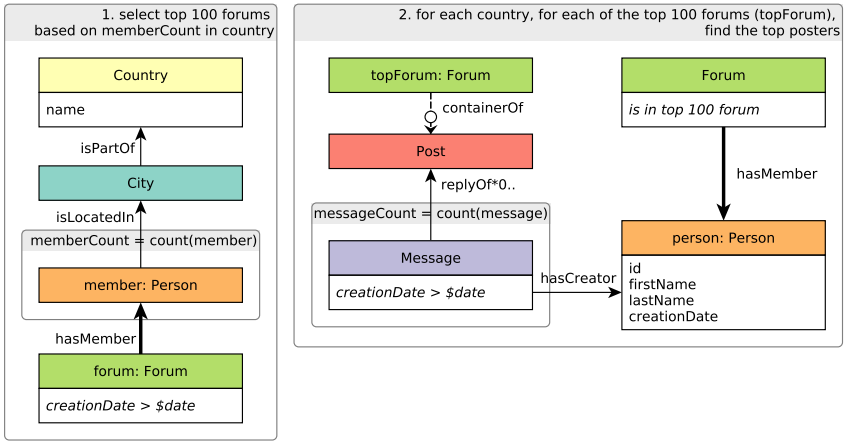
sort

1	messageCount	↓	
2	forum.id	↑	

limit	20
CPs	1.1, 1.2, 1.3, 2.1, 2.2, 2.4, 3.3, 8.2

BI / read / 4

BI 1 BI 2 BI 3 BI 4 BI 5 BI 6 BI 7 BI 8 BI 9 BI 10 BI 11 BI 12 BI 13 BI 14 BI 15 BI 16 BI 17 BI 18 BI 19 BI 20

query	BI / read / 4
title	Top message creators by country
pattern	
desc.	Find the most popular Forums by Country, where the popularity of a Forum is measured by the number of members that Forum has from a given Country. Calculate the top 100 most popular Forums. If a Forum is popular in multiple countries, it should only be calculated once with its largest membership. In case of a tie, the Forum(s) with the smaller id value(s) should be selected. For each member Person of the 100 most popular Forums, count the number of Messages (`messageCount`) they made in any of those (most popular) Forums. Also include those member Persons who have not posted any Messages (have a `messageCount` of 0).

params

1	`date`	Date	Selected from the first 30 days of the network

result

1	`person.id`	ID	R	
2	`person.firstName`	String	R	
3	`person.lastName`	String	R	
4	`person.creationDate`	DateTime	R	
5	`messageCount`	32-bit Integer	A	

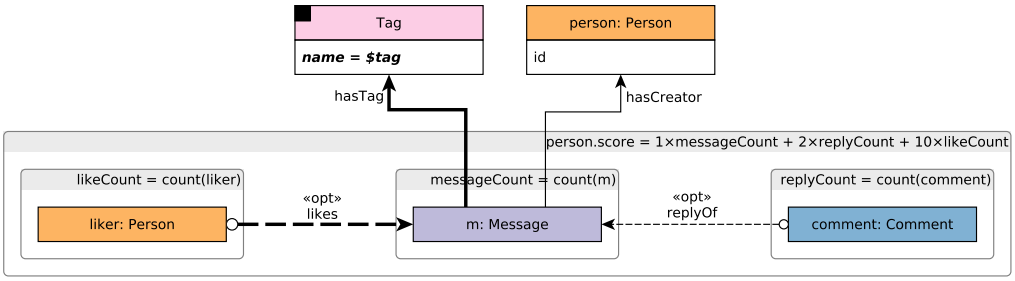
sort

1	`messageCount`	↓	
2	`person.id`	↑	

limit	100
CPs	1.2, 1.3, 2.1, 2.2, 2.3, 2.4, 3.3, 5.3, 6.1, 8.2, 8.4

BI / read / 5

BI 1 BI 2 BI 3 BI 4 BI 5 BI 6 BI 7 BI 8 BI 9 BI 10 BI 11 BI 12 BI 13 BI 14 BI 15 BI 16 BI 17 BI 18 BI 19 BI 20

query	BI / read / 5
title	Most active posters of a given topic
pattern	
desc.	Get each Person (`person`) who has created a Message (`message`) with a given Tag (direct relation, not transitive). Considering only these Messages, for each Person node: • Count its `messages` (`messageCount`). • Count likes (`likeCount`) to its `messages`. • Count Comments (`replyCount`) in reply to it `messages`. The `score` is calculated according to the following formula: $1 \times$ `messageCount` $+ 2 \times$ `replyCount` $+ 10 \times$ `likeCount`.

params

1	`tag`	Long String	Tags with a similar amount of Messages are selected

result

1	`person.id`	ID	R	
2	`replyCount`	32-bit Integer	A	
3	`likeCount`	32-bit Integer	A	
4	`messageCount`	32-bit Integer	A	
5	`score`	32-bit Integer	A	

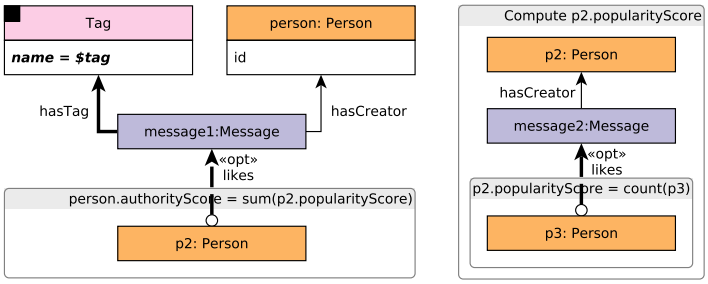
sort

1	`score`	↓	
2	`person.id`	↑	

limit	100
CPs	1.2, 2.3, 8.2

BI / read / 6

BI 1
BI 2
BI 3
BI 4
BI 5
BI 6
BI 7
BI 8
BI 9
BI 10
BI 11
BI 12
BI 13
BI 14
BI 15
BI 16
BI 17
BI 18
BI 19
BI 20

query	BI / read / 6
title	Most authoritative users on a given topic
pattern	
desc.	Given a Tag (tag), find all Persons (person) that ever created a Message with the Tag. For each of these Persons (person) compute their "authority score" as follows: • The "authority score" is the sum of "popularity scores" of the Persons (p2) that liked any of that Person's Messages with the given Tag (same criterion as for message1). • A Person's (p2) "popularity score" is defined as the total number of likes on all of their Messages (message2).

params

1	tag	Long String	Tags with a similar amount of Messages are selected

result

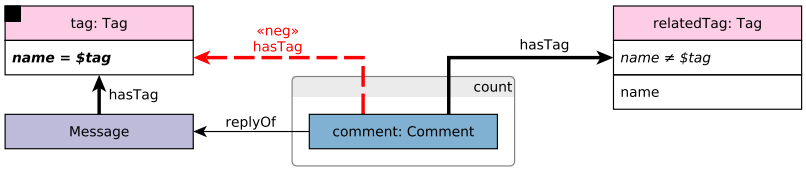
1	person.id	ID	R	
2	authorityScore	32-bit Integer	A	

sort

1	authorityScore	↓
2	person1.id	↑

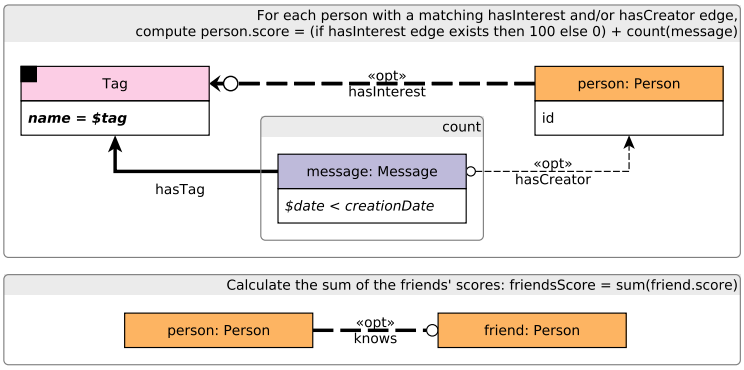
limit	100
CPs	1.2, 2.3, 3.3, 6.1, 8.2
relevance	Computing the authority scores might involve computing the popularity score for the same Person multiple times. Implementations are advised to avoid such redundant computations.

BI / read / 7

query	BI / read / 7
title	Related topics
pattern	
desc.	Find all Messages that have a given Tag. Find the related Tags attached to (direct) reply Comments of these Messages, but only of those reply Comments that do not have the given Tag. Group the Tags by name, and get the count of replies in each group.
params	1 | tag | Long String | Tags with a similar amount of Messages are selected
result	1 | relatedTag.name | Long String | R | 2 | count | 32-bit Integer | A |
sort	1 | count | ↓ | 2 | relatedTag.name | ↑ |
limit	100
CPs	1.4, 3.3, 5.2, 8.1

BI / read / 8

BI 1 BI 2 BI 3 BI 4 BI 5 BI 6 BI 7 BI 8 BI 9 BI 10 BI 11 BI 12 BI 13 BI 14 BI 15 BI 16 BI 17 BI 18 BI 19 BI 20

query	BI / read / 8
title	Central person for a tag
pattern	
desc.	Given a Tag, find all Persons that are interested in the Tag and/or have written a Message (Post or Comment) with a `creationDate` after a given `date` and that has a given Tag. For each Person, compute the `score` as the sum of the following two aspects: • 100, if the Person has this Tag as their interest, or 0 otherwise • number of Messages by this Person with the given Tag Also, for each Person, compute the sum of the score of the Person's friends (`friendsScore`).

params

1	tag	Long String	Tags with a similar amount of Messages are selected
2	date	Date	Dates from around the same day are selected. (TODO - how exactly? what distribution?)

result

1	person.id	ID	R	
2	score	32-bit Integer	A	
3	friendsScore	32-bit Integer	A	The sum of the score of the person's friends

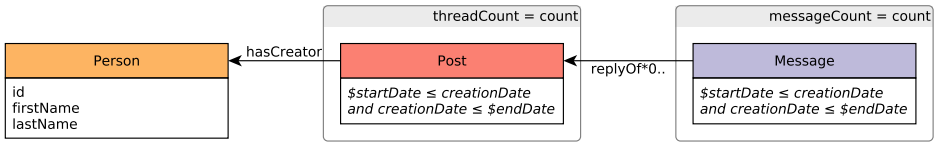
sort

1	score + friendsScore	↓
2	person.id	↑

limit	100
CPs	1.2, 2.1, 2.3, 3.2, 5.3, 8.2, 8.4, 8.5
relevance	Similarly to BI 16, there are two major ways to compute this query: (1) creating an induced subgraph of the interested Persons and their friends and performing the scoring on this graph or (2) performing the scoring without creating an induced subgraph and scoring the friends of a Person on-the-fly. The first approach is more efficient as it avoids redundant computations, however, specifying it needs support for composable graph queries.

BI / read / 9

BI 1 BI 2 BI 3 BI 4 BI 5 BI 6 BI 7 BI 8 BI 9 BI 10 BI 11 BI 12 BI 13 BI 14 BI 15 BI 16 BI 17 BI 18 BI 19 BI 20

query	BI / read / 9
title	Top thread initiators
pattern	
desc.	For each Person, count the number of Posts they created in the time interval `[startDate, endDate)` (equivalent to the number of threads they initiated) and the number of Messages in each of their (transitive) reply trees, including the root Post of each tree. When calculating Message counts only consider Messages created within the given time interval. Return each Person, number of Posts they created, and the count of all Messages that appeared in the reply trees (including the Post at the root of tree) they created.

params

1	`startDate`	Date	TODO
2	`endDate`	Date	8-10 days after the `startDate`

result

1	`person.id`	ID	R	
2	`person.firstName`	String	R	
3	`person.lastName`	String	R	
4	`threadCount`	32-bit Integer	A	The number of Posts created by that Person (the number of threads initiated)
5	`messageCount`	32-bit Integer	A	The number of Messages created in all the threads this Person initiated

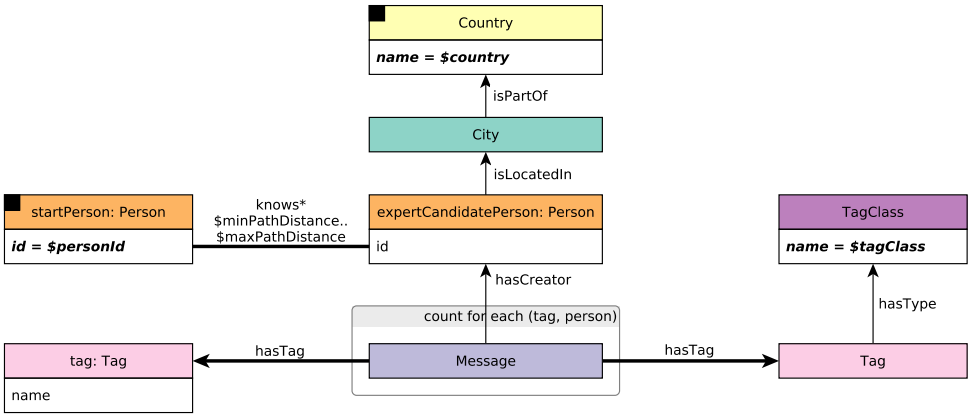
sort

1	`messageCount`	↓
2	`person.id`	↑

limit	100
CPs	1.2, 2.2, 2.3, 3.2, 7.2, 7.3, 7.4, 8.1, 8.5

BI / read / 10

BI 1 BI 2 BI 3 BI 4 BI 5 BI 6 BI 7 BI 8 BI 9 BI 10 BI 11 BI 12 BI 13 BI 14 BI 15 BI 16 BI 17 BI 18 BI 19 BI 20

query	BI / read / 10
title	Experts in social circle
pattern	
desc.	Given a Person (`startPerson`), find all other Persons (`expertCandidatePerson`) that live in a given Country and are connected to given Person by a *shortest path* with length in range `[minPathDistance, maxPathDistance]` through the knows relation. For each of these `expertCandidatePerson` nodes, retrieve all of their Messages that contain at least one Tag belonging to a given TagClass (direct relation not transitive). For each Message, retrieve all of its Tags. Group the results by Persons and Tags, then count the Messages by a certain Person having a certain Tag.

params

1	`personId`	ID	The ID of the `startPerson`. Persons with a similar degree of knows edges are selected
2	`country`	String	Countries with a similar number of Persons are selected
3	`tagClass`	Long String	TagClasses with a similar degree of hasType edges are selected
4	`minPathDistance`	32-bit Integer	1 or 2
5	`maxPathDistance`	32-bit Integer	2 or 3

result

1	`expertCandidatePerson.id`	ID	R	
2	`tag.name`	Long String	R	
3	`messageCount`	32-bit Integer	A	Number of Messages created by that Person containing that Tag

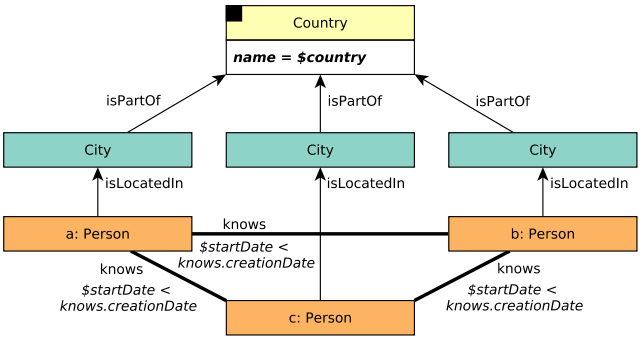
sort

1	`messageCount`	↓
2	`tag.name`	↑
3	`expertCandidatePerson.id`	↑

limit	100
CPs	1.2, 1.3, 2.3, 2.4, 3.3, 5.3, 7.1, 7.2, 7.3, 8.1, 8.6

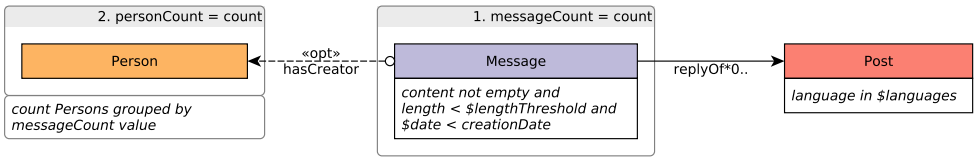
BI / read / 11

BI 1 BI 2 BI 3 BI 4 BI 5 BI 6 BI 7 BI 8 BI 9 BI 10 BI 11 BI 12 BI 13 BI 14 BI 15 BI 16 BI 17 BI 18 BI 19 BI 20

query	BI / read / 11
title	Friend triangles
pattern	
desc.	For a given `country`, count all the distinct triples of Persons such that: • `a` is friend of `b`, • `b` is friend of `c`, • `c` is friend of `a`, and these friendships were created after a given `startDate`. Distinct means that given a triple t_1 in the result set R of all qualified triples, there is no triple t_2 in R such that t_1 and t_2 have the same set of elements.
params	1 `country` Long String 2 `startDate` Date
result	1 `count` 32-bit Integer A
limit	n/a
CPs	1.1, 2.3, 2.5

BI / read / 12

BI 1 BI 2 BI 3 BI 4 BI 5 BI 6 BI 7 BI 8 BI 9 BI 10 BI 11 BI 12 BI 13 BI 14 BI 15 BI 16 BI 17 BI 18 BI 19 BI 20

query	BI / read / 12
title	How many persons have a given number of messages
pattern	
desc.	For each Person, count the number of Messages they made (`messageCount`). Only count Messages with the following attributes: • Its `content` is not empty (and consequently, the `imageFile` attribute is empty for Posts). • Its `length` is below the `lengthThreshold` (exclusive, equality is not allowed). • Its `creationDate` is after `date` (exclusive, equality is not allowed). • It is written in any of the given `languages`. – The language of a Post is defined by its `language` attribute. – The language of a Comment is that of the Post that initiates the thread where the Comment replies to. The Post and Comments in the reply tree's path (from the Message to the Post) do not have to satisfy the constraints for `content`, `length` and `creationDate`. For each `messageCount` value, count the number of Persons with exactly `messageCount` Messages (with the required attributes).

params

1	`date`	Date	
2	`lengthThreshold`	32-bit Integer	Selected as balanced against `date` to filter around 30% of the Messages within a language and keep the variance low
3	`languages`	{String}	Only the most frequently used languages are selected

result

1	`messageCount`	32-bit Integer	A	Number of Messages created
2	`personCount`	32-bit Integer	A	Number of Persons with `messageCount` Messages

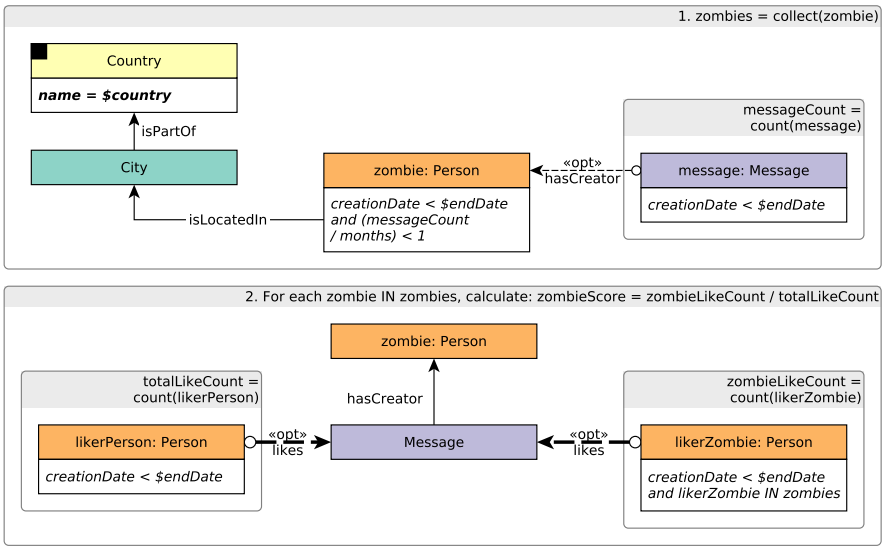
sort

1	`personCount`	↓
2	`messageCount`	↓

limit	n/a
CPs	1.1, 1.2, 1.4, 3.2, 4.2, 4.3, 8.1, 8.2, 8.3, 8.4, 8.5

BI / read / 13

BI 1 BI 2 BI 3 BI 4 BI 5 BI 6 BI 7 BI 8 BI 9 BI 10 BI 11 BI 12 BI 13 BI 14 BI 15 BI 16 BI 17 BI 18 BI 19 BI 20

query	BI / read / 13
title	Zombies in a country
pattern	
desc.	Find zombies within the given `country`, and return their zombie scores. A `zombie` is a Person created before the given `endDate`, which has created an average of `[0, 1)` Messages per month, during the time range between profile's `creationDate` and the given `endDate`. The number of months spans the time range from the `creationDate` of the profile to the `endDate` with partial months on both end counting as one month (e.g. a `creationDate` of Jan 31 and an `endDate` of Mar 1 result in 3 months). For each `zombie`, calculate the following: • `zombieLikeCount`: the number of likes received from other zombies. • `totalLikeCount`: the total number of likes received. • `zombieScore`: `zombieLikeCount` / `totalLikeCount`. If the value of `totalLikeCount` is 0, the `zombieScore` of the `zombie` should be 0.0. For both `zombieLikeCount` and `totalLikeCount`, only consider likes received from profiles that were created before the given `endDate`.

params

1	country	Long String	Only the largest Countries are selected
2	endDate	Date	Selected from the last days of the initial data set

result

1	zombie.id	ID	R	
2	zombieLikeCount	32-bit Integer	A	
3	totalLikeCount	32-bit Integer	A	
4	zombieScore	32-bit Float	A	Determined as `zombieLikeCount` / `totalLikeCount`

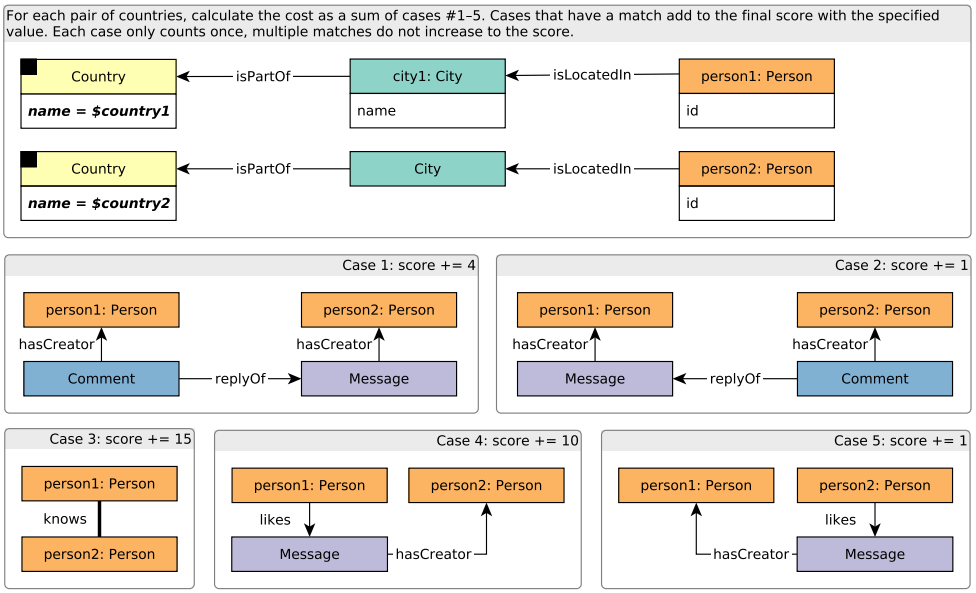
sort

1	zombieScore	↓
2	zombie.id	↑

limit	100
CPs	1.2, 2.1, 2.3, 2.4, 3.2, 3.3, 4.2, 5.1, 5.3, 8.2, 8.4, 8.5

BI / read / 14

BI 1 BI 2 BI 3 BI 4 BI 5 BI 6 BI 7 BI 8 BI 9 BI 10 BI 11 BI 12 BI 13 BI 14 BI 15 BI 16 BI 17 BI 18 BI 19 BI 20

query	BI / read / 14
title	International dialog
pattern	
desc.	Consider all pairs of people (`person1`, `person2`) such that one is located in a City of Country `country1` and the other is located in a City of Country `country2`. For each City of Country `country1`, return the highest scoring pair. The score of a pair is defined as the sum of the subscores awarded for the following kinds of interaction. The initial value is `score = 0`. 1. `person1` has created a reply Comment to at least one Message by `person2`: `score += 4` 2. `person1` has created at least one Message that `person2` has created a reply to: `score += 1` 3. `person1` and `person2` know each other: `score += 15` 4. `person1` liked at least one Message by `person2`: `score += 10` 5. `person1` has created at least one Message that was liked by `person2`: `score += 1` Consequently, the maximum score a pair can obtain is: `4 + 1 + 15 + 10 + 1 = 31`. This query has two variants based on whether the input parameters are selected as correlated (close countries) or uncorrelated (far countries).

params

1	country1	Long String	A: correlated with parameter country2, i.e. the countries are close and there are many Persons visiting both Countries. B: uncorrelated with parameter country2, i.e. the countries are afar and there are few Persons visiting both Countries.
2	country2	Long String	

result

1	person1.id	ID	R	
2	person2.id	ID	R	
3	city1.name	Long String	R	
4	score	32-bit Integer	C	

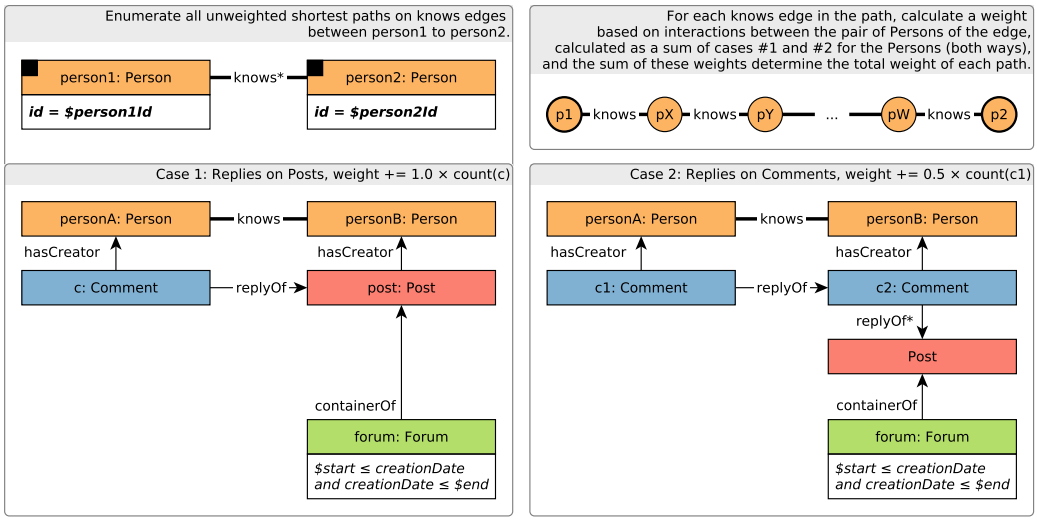
sort

1	score	↓	
2	person1.id	↑	
3	person2.id	↑	

limit	n/a
CPs	1.3, 1.4, 2.1, 3.1, 3.3, 5.1, 5.2, 5.3, 8.3, 8.4

BI / read / 15

BI 1
BI 2
BI 3
BI 4
BI 5
BI 6
BI 7
BI 8
BI 9
BI 10
BI 11
BI 12
BI 13
BI 14
BI 15
BI 16
BI 17
BI 18
BI 19
BI 20

query	BI / read / 15
title	Trusted connection paths through forums created in a given timeframe
pattern	
desc.	Given two Persons, find all (unweighted) shortest paths between these two Persons, in the subgraph induced by the knows relationship. Then, for each path calculate a weight. The nodes in the path are Persons, and the weight of a path is the sum of weights between every pair of consecutive Person nodes in the path. The weight for a pair of Persons is calculated based on their interactions: • Every direct reply (by one of the Persons) to a Post (by the other Person) contributes 1.0. • Every direct reply (by one of the Persons) to a Comment (by the other Person) contributes 0.5. Only consider Messages that were created in a Forum that was created within the timeframe (interval) `[startDate, endDate)`. Note that for Comments, the containing Forum is that of the Post that the comment (transitively) replies to. Also note that interactions are counted both ways. Return all paths with the Person IDs ordered by their weights descending.

params

1	`person1Id`	ID	
2	`person2Id`	ID	
3	`startDate`	Date	
4	`endDate`	Date	

result

1	`person.id`	[ID]	C	Ordered sequence of the Person IDs in the path
2	`weight`	32-bit Float	C	

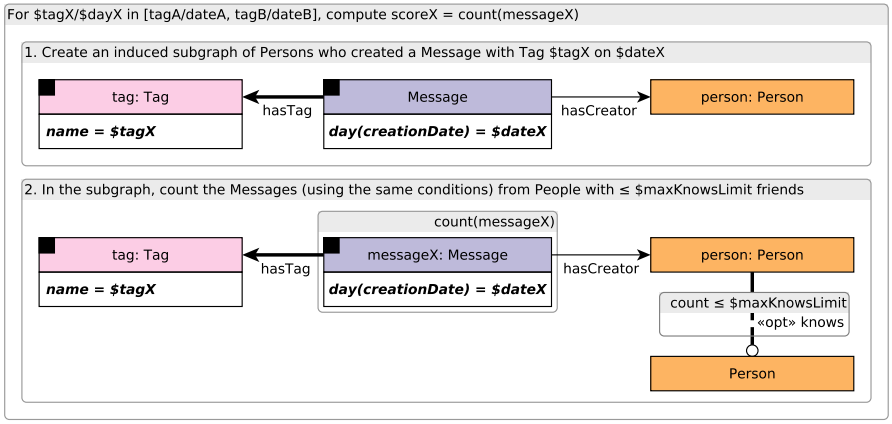
sort

1	`weight`	↓	The order of paths with the same weight is unspecified
2	`personIds`	↑	The IDs in the paths are used for lexicographical sorting

limit	n/a
CPs	1.2, 2.1, 2.2, 2.4, 3.3, 5.1, 5.3, 7.2, 7.3, 7.5, 7.7, 8.1, 8.2, 8.3, 8.4, 8.5, 8.6

BI / read / 16

BI 1 BI 2 BI 3 BI 4 BI 5 BI 6 BI 7 BI 8 BI 9 BI 10 BI 11 BI 12 BI 13 BI 14 BI 15 BI 16 BI 17 BI 18 BI 19 BI 20

query	BI / read / 16
title	Fake news detection
pattern	
desc.	Given two Tag/date pairs (`tagA`/`dateA` and `tagB`/`dateB`), for each pair `tagX`/`dateX`: • Create an induced subgraph between Persons where for each pair of Persons `person1`/`person2`, both have created a Message on the day of `dateX` with Tag `tagX`. • In the induced subgraph, only keep pairs of Persons who have at most `maxKnowsLimit` friends (in the induced subgraph). • For these Persons, count the number of Messages created on `dateX` with Tag `tagX`. Return Persons who had at least one Messages for both `tagA`/`dateA` and `tagB`/`dateB` ranked by their total number of Messages (descending).

params

1	`tagA`	Long String	
2	`dateA`	Date	
3	`tagB`	Long String	
4	`dateB`	Date	
5	`maxKnowsLimit`	32-bit Integer	Selected between 3 and 6

result

1	`person.id`	ID	R	
2	`messageCountA`	32-bit Integer	A	Message count for `tagA`/`dateA`
3	`messageCountB`	32-bit Integer	A	Message count for `tagB`/`dateB`

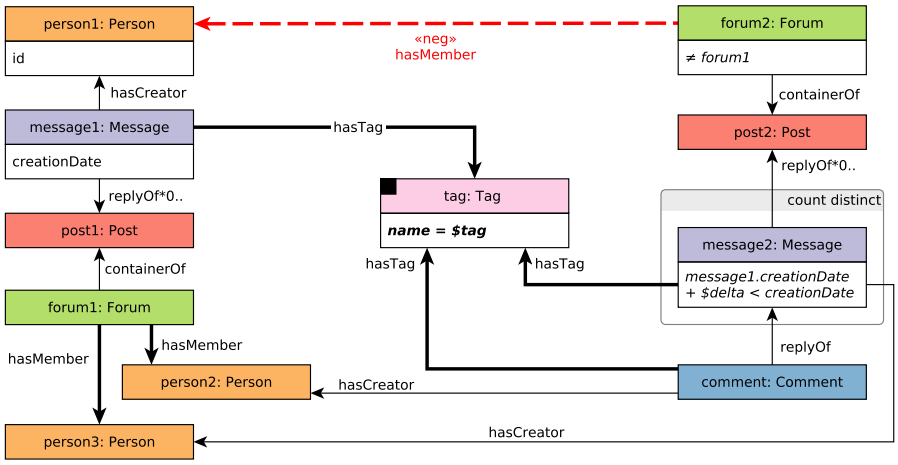
sort

1	`messageCountA` + `messageCountB`	↓
2	`person.id`	↑

limit	20
CPs	5.3, 8.4, 8.5
relevance	There are two major ways to compute this query: (1) create the induced subgraph as suggested by the specification (either as a view or in materialized form), or (2) skip creating the induced subgraph and perform on-the-fly check for the number of friends (who also posted at least one Message with the given Tag on the given date). The latter approach is easier to express in systems which do not provide graph views but might result in redundant computations (the query engine will might repeatedly check whether a Person has at least one Message that satifies the conditions).

BI / read / 17

BI 1
BI 2
BI 3
BI 4
BI 5
BI 6
BI 7
BI 8
BI 9
BI 10
BI 11
BI 12
BI 13
BI 14
BI 15
BI 16
BI 17
BI 18
BI 19
BI 20

query	BI / read / 17
title	Information propagation analysis
pattern	
desc.	This query aims to identify instances of "information propagation" when a Person (`person1`) submits a Message (`message1`) with a given Tag (`tag`) to a Forum (`forum1`). This is read by other members of `forum1`, Persons `person2` and `person3`. Some time later (specified by the `delta` parameter), these persons have a discussion with the same `tag` in a different Forum (`forum2`) where `person1` is not a member. The discussion consists of a Message (`message2`) by `person2` and a direct reply Comment (`comment`) by `person3`. Return IDs of `person1` with the number of interactions their Messages (might have) caused.

params

1	tag	Long String	Tags with a similar amount of Messages are selected
2	delta	32-bit Integer	Measured in hours, selected to be between 8 and 16 hours.

result

1	person1.id	ID	R
2	messageCount	32-bit Integer	A

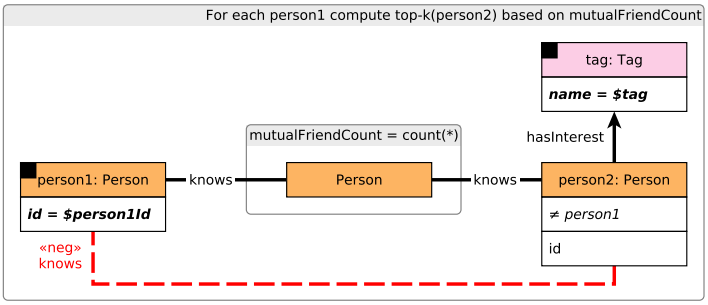
sort

1	messageCount	↓
2	person1.id	↑

limit	10
CPs	2.1, 2.3, 8.1

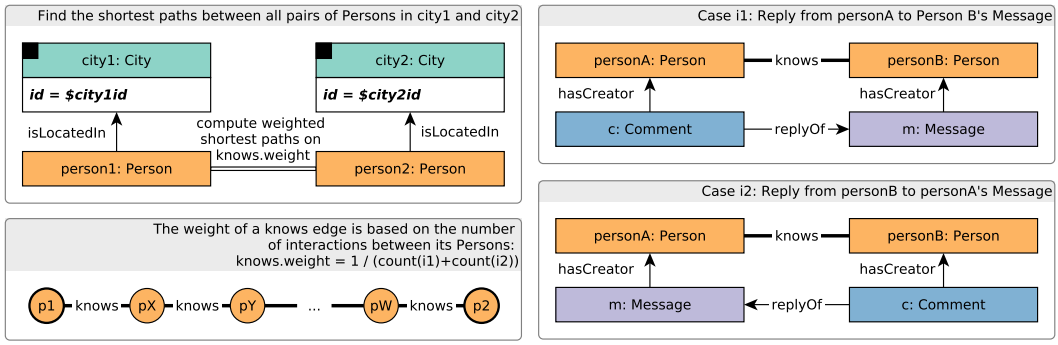
BI / read / 18

BI 1
BI 2
BI 3
BI 4
BI 5
BI 6
BI 7
BI 8
BI 9
BI 10
BI 11
BI 12
BI 13
BI 14
BI 15
BI 16
BI 17
BI 18
BI 19
BI 20

query	BI / read / 18
title	Friend recommendation
pattern	
desc.	For a given Person (`person1`) and a Tag (`tag`), recommend new friends (`person2`) who • do not yet know `person1` • have many mutual friends with `person1` • are interested in `tag`. Rank Persons `person2` based on the number of mutual friends.
params	1 `person1Id` ID — Persons with a similar amount of friends are selected 2 `tag` Long String — Tags with a similar amount of Messages are selected
result	1 `person2.id` ID R 2 `mutualFriendCount` 32-bit Integer A
sort	1 `mutualFriendCount` ↓ 2 `person2.id` ↑
limit	20
CPs	2.5, 8.1

BI / read / 19

BI 1
BI 2
BI 3
BI 4
BI 5
BI 6
BI 7
BI 8
BI 9
BI 10
BI 11
BI 12
BI 13
BI 14
BI 15
BI 16
BI 17
BI 18
BI 19
BI 20

query	BI / read / 19
title	Interaction path between cities
pattern	
desc.	Given two Cities `city1`, `city2`, find Persons `person1`, `person2` living in these Cities (respectively) with the shortest *interaction path* between them. If there are multiple pairs of people with shortest paths having the same total weight, return all of them. The shortest path is computed using a weight between two Persons defined as the reciprocal of the number of interactions (direct reply Comments to a Message by the other Person). Therefore, more interactions imply a smaller weight. *Note:* Interactions are counted both ways, i.e. if Alice writes 2 reply Comments to Bob's Messages and Bob writes 3 reply Comments to Alice's Messages, their total number of interactions is 5.

params

1	`city1Id`	ID	Small Cities within the same Country are selected
2	`city2Id`	ID	

result

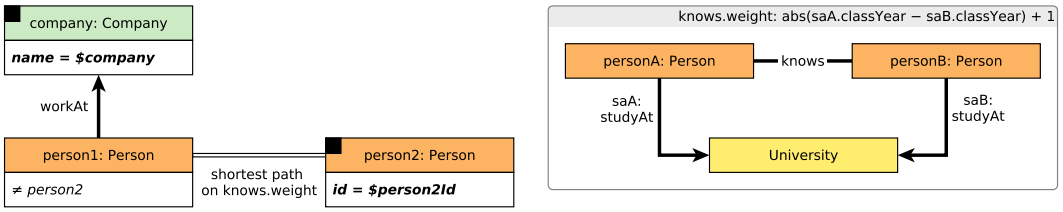
1	`person1.id`	ID	R
2	`person2.id`	ID	R
3	`totalWeight`	32-bit Float	C

sort

1	`totalWeight`	↑
2	`person1.id`	↑
3	`person2.id`	↑

limit	20
CPs	3.3, 7.6, 7.7, 8.4, 8.6
relevance	Finding shortest paths between pairs of Persons in Cities can be implemented in theory with an *all-pairs shortest paths* algorithm. However, this needs to be executed on the whole Person-knows-Person graph (with edge weights derived from the number of interactions) so it is expected to be prohibitively expensive. A better approach is using multiple *single-source shortest path algorithms* (e.g. from the City with fewer inhibitants).

BI / read / 20

query	BI / read / 20
title	Recruitment
pattern	
desc.	Given a Company `company` and a Person `person2` (who is known to be working at another Company), find a different Person (`person1`) who works in `company` and is reachable by from `person2` through people who have studied together. On this path, we only consider edges between Persons who know each other and attended the same university and set the weight of the edge to the absolute difference between the year of enrolment plus 1 (`studyAt.classYear + 1`). We return the 20 shortest paths. If there are multiple Person `person1` nodes with the same shortest path, return all of them.

params

1	company	Long String	Companies with a similar number of employees (former or current) are selected
2	person2Id	ID	

result

1	person1.id	ID	R	
2	totalWeight	64-bit Integer	C	

sort

1	totalWeight	↑	
2	person1.id	↑	

limit	20
CPs	3.3, 7.6, 7.7, 8.4, 8.6
relevance	Implementations can either pre-compute edge weights or compute them on-the-fly. To find a weighted shortest path efficiently, implementations can use e.g. a bidirectional Dijkstra algorithm.

6.2 Refreshes

6.2.1 Inserts

See *Interactive Inserts* (Section 5.3).

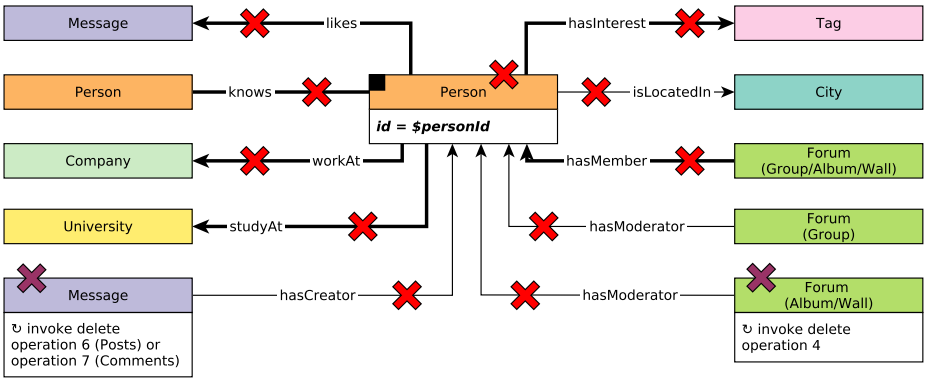
6.2.2 Deletes

Each delete query removes

1. a single edge between two existing nodes
2. or a node, all incident edges and, in certain cases, nodes and edges that are transitively reachable on a certain path.

BI / delete / 1

DEL 1
DEL 2
DEL 3
DEL 4
DEL 5
DEL 6
DEL 7
DEL 8

query	BI / delete / 1
title	Remove person and its personal forums and message (sub)threads
pattern	
desc.	Remove a Person and its edges (isLocatedIn, studyAt, workAt, hasInterest, likes, knows, hasMember, hasModerator, hasCreator). Additionally, remove the Album and Wall Forums whose moderator is the Person and remove all Messages the Person has created in the rest of the Forums (Groups).
params	1 personId ID
CPs	9.3, 9.4, 9.5
relevance	• Removal of a Person removes Forums of type "Walls" and "Albums" but not "Groups", which can continue if even the founder has left the network. For Groups, the hasModerator edge is deleted. We have discussed various approaches to appoint a new moderator, e.g. 1. choose member at random from the set of existing group members or 2. the member with the oldest group join date becomes the moderator. However, to keep the generator and the workload simple, currently no moderator is selected, leaving the group without a moderator. • Removal of a Person removes all Posts/Comments they are creator of this could result in the removal of a Comment in the middle of a thread.

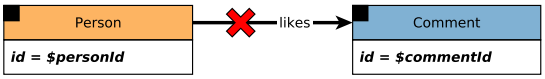
BI / delete / 2

DEL 1
DEL 2
DEL 3
DEL 4
DEL 5
DEL 6
DEL 7
DEL 8

query	BI / delete / 2
title	Remove post like
pattern	
desc.	Given a Person and a Post, remove the likes edge between them.
params	1 personId ID 2 postId ID
CPs	9.4
relevance	Removal of a likes edge is a rare event, e.g. people accidently liking a Post, this can be reflected by the relative frequency of the operation.

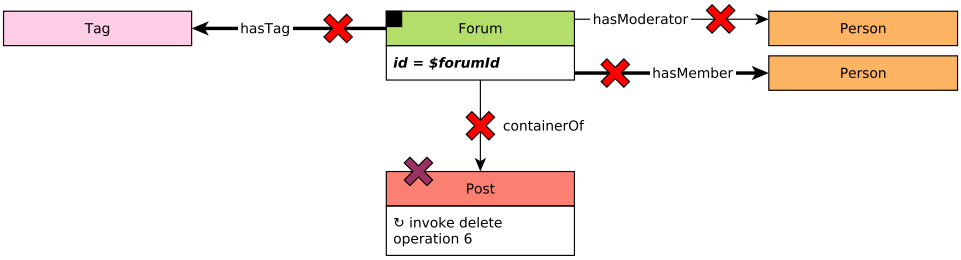
BI / delete / 3

DEL 1
DEL 2
DEL 3
DEL 4
DEL 5
DEL 6
DEL 7
DEL 8

query	BI / delete / 3
title	Remove comment like
pattern	
desc.	Given a Person and a Comment, remove the likes edge between them.
params	1 personId ID 2 commentId ID
CPs	9.4
relevance	Removal of a likes edge is a rare event, e.g. people accidently liking a Comment, this can be reflected by the relative frequency of the operation.

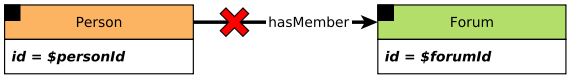
BI / delete / 4

DEL 1
DEL 2
DEL 3
DEL 4
DEL 5
DEL 6
DEL 7
DEL 8

query	BI / delete / 4
title	Remove forum and its content
pattern	
desc.	Remove a Forum and its edges (hasModerator, hasMember, hasTag) and all Posts in the Forum (connected by containerOf edges) and their direct and transitive Comments.
params	1 forumId ID
CPs	9.3, 9.4, 9.5
relevance	n/a

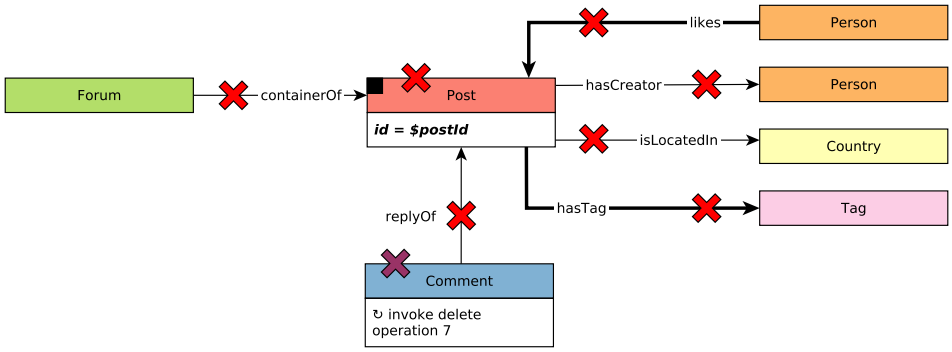
BI / delete / 5

DEL 1
DEL 2
DEL 3
DEL 4
DEL 5
DEL 6
DEL 7
DEL 8

query	BI / delete / 5
title	Remove forum membership
pattern	
desc.	Given a Forum and a Person, remove the hasMember edge between them.
params	1 forumId ID 2 personId ID
CPs	9.4
relevance	n/a

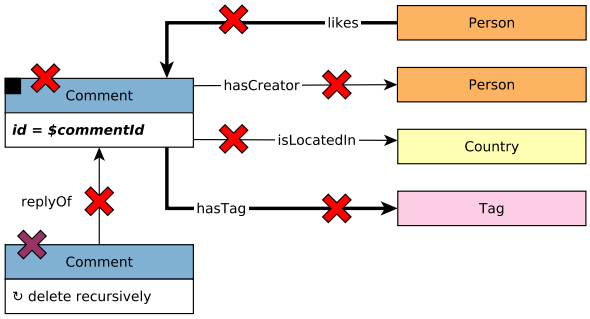
BI / delete / 6

DEL 1
DEL 2
DEL 3
DEL 4
DEL 5
DEL 6
DEL 7
DEL 8

query	BI / delete / 6
title	Remove post thread
pattern	
desc.	Remove a Post node and its edges (isLocatedIn, likes, hasCreator, hasTag, containerOf). Remove all replies to the Post and the connecting replyOf edges. In addition, remove all transitive reply Comments to the Post and their edges.
params	1 postId ID
CPs	9.3, 9.4, 9.5
relevance	n/a

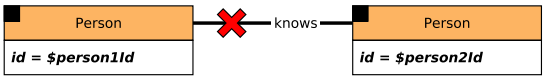
BI / delete / 7

DEL 1
DEL 2
DEL 3
DEL 4
DEL 5
DEL 6
DEL 7
DEL 8

query	BI / delete / 7
title	Remove comment subthread
pattern	
desc.	Remove a Comment node and its *edges* (isLocatedIn, likes, hasCreator, hasTag). In addition, remove all replies to the Comment connected by replyOf and their *edges*.
params	1 commentId ID
CPs	9.3, 9.4, 9.5
relevance	n/a

BI / delete / 8

DEL 1
DEL 2
DEL 3
DEL 4
DEL 5
DEL 6
DEL 7
DEL 8

query	BI / delete / 8
title	Remove friendship
pattern	
desc.	Given two Person nodes, remove the knows edge between them.
params	1 person1Id ID 2 person2Id ID
CPs	9.4
relevance	n/a

7 Auditing Policies

This chapter contains the auditing policies for the LDBC Social Network Benchmark. The initial draft of the auiting policies were published in the EU project deliverable D6.3.3 "LDBC Benchmark Auditing Policies" [23].

This chapter is divided in the following parts:

- Motivation of benchmark result auditing
- General discussion of auditable aspects of benchmarks
- Specific checklists and running rules for the Social Network Benchmark (SNB)

Many definitions and general considerations are shared between the benchmarks, hence it is justified to present the principles first and to refer to these in the context of the benchmark specific rules.

7.1 Rationale and General Principles

The purpose of benchmark auditing is to improve the *credibility* and *reproducibility* of benchmark claims by involving a set of detailed execution rules and third party verification of compliance with these.

Rules may exist separately of auditing but auditing is not meaningful unless the rules are adequately precise. Aspects like auditor training and qualification cannot be addressed separately from a discussion of the matters the auditor is supposed to verify. Thus the credibility of the entire process hinges on clear and shared understanding of what a benchmark is expected to demonstrate and on the auditor being capable of understanding the process and of verifying that the benchmark execution is fair and does not abuse the rules or pervert the objectives of the benchmark.

Due to the open-ended nature of technology and the agenda of furthering innovation via measurement, it is not feasible or desirable to over-specify the limits of benchmark implementation. Hence there will always remain judgement calls for borderline cases. In this respect auditing and the LDBC are not separate. It is expected that issues of compliance as well as of maintenance of rules will come before the LDBC as benchmark claims are made.

7.2 Auditing Rules Overview

7.2.1 Auditor Training, Certification, and Selection

7.2.1.1 Auditor Training

Auditor training consists of familiarisation with the benchmark and existing implementations thereof. This involves the auditor candidate running the reference implementations of the benchmark in order to see what is normal behaviour and practice in the workload. The training and practice may involve communication with the benchmark task force for clarifying intent and details of the benchmark rules. This produces feedback for the task force for further specification of the rules.

7.2.1.2 Auditor Certification

The auditor certification and qualification is done in the form of an examination administered by the task force responsible for the benchmark being audited. The examination may be carried out by teleconference. The task force will subsequently vote on accepting each auditor, by simple majority. An auditor is certified for a particular benchmark by the task force maintaining the benchmark in question.

7.2.1.3 Auditor Selection

In the default auditor selection, the task force responsible for the benchmark being audited appoints a third party, impartial auditor. The task force may in special cases appoint itself as auditor of a particular result. This is not, however, the preferred course of action but may be done if no suitable third party auditor is available

7.2.2 Auditing Process Stages

7.2.2.1 Performing a Benchmark Audit

A benchmark result is to be audited by an LDBC appointed auditor or the LDBC task force managing the benchmark. An LDBC audit may be performed by remote login and does not require the auditor's physical presence on site. The test sponsor shall grant the auditor any access necessary for validating the benchmark run. This will typically include administrator access to the SUT hardware.

7.2.2.2 Benchmark-Specific Checklist

Each benchmark specifies a checklist to be verified by the auditor. The benchmark run shall be performed by the auditor. The auditor shall take copies of relevant configuration files and test results for future checking and insertion into the full disclosure report.

7.2.2.3 Producing the FDR

The FDR is produced by the auditor or auditors, with any required input from the test sponsor. Each non-default configuration parameter needs to be included in the FDR and justification needs to be provided why the given parameter was changed. The auditor produces an attestation letter that verifies authenticity of the presented results. This letter is to be included into the FDR as an addendum. The attestation letter has no specific format requirements but shall state that the auditor has established compliance with a specified version of the benchmark specification.

7.2.2.4 Publishing the FDR

The FDR and any benchmark specific summaries thereof shall be published on the LDBC website.

7.2.3 Other Procedures

7.2.3.1 Challenges

A benchmark result may be challenged for non-compliance with LDBC rules. The benchmark task force responsible for maintenance of the benchmark will rule on matters of compliance. A result found to be non-compliant will be withdrawn from the list of official LDBC benchmark results.

7.3 Auditable Properties of Systems and Benchmark Implementations

7.3.1 Validation of Query Results and ACID Properties

7.3.1.1 Correctness of Results

A benchmark should be published with a deterministically reproducible validation dataset. Validation queries applied to the validation dataset will deterministically produce a set of correct answers. This is used in the first stage of benchmark run to test for the correctness of an SUT or benchmark implementation. This validation stage is not timed.

Inputs for validation The validation takes the form of a set of data generator parameters, a set of test queries that at least include one instance of each of the workload query templates and the expected results.

Approximate results and error margin In certain cases the results may be approximate. This may happen in cases of non-unique result ordering keys, imprecise numeric data types, random behaviours in certain graph analytics algorithms etc. Therefore, a validation set shall specify the degree of allowable error: For example, for counts, the value must be exact, for sums, averages and the like, at least 8 significant digits are needed, for statistical measures like graph centralities, the result must be within 1% of the reference result. Each benchmark shall specify its expectation in an unambiguously verifiable manner.

7.3.1.2 ACID Compliance

As part of the auditing process, the auditors ascertain that the SUT satisfies "the ACID properties", i.e. it provides atomic transactions, complies with its claimed isolation level, and ensures durability in case of failures. This section outlines transactional behaviours of SUTs which are checked in the course of auditing an SUT in a given benchmark.

A benchmark specifies transactional semantics that may be required for different parts of the workload. The requirements will typically be different for initial bulk load of data and for the workload itself. Different sections of the workload may further be subject to different transactionality requirements.

No finite series of tests can prove that the ACID properties are fully supported. Passing the specified tests is a necessary, but not sufficient, condition for meeting the ACID requirements. However, for fairness of reporting, only the tests specified here are required and must appear in the FDR for a benchmark. (This is taken exactly from the TPC-C specification [64].)

The properties for ACID compliance are defined as follows:

Atomicity Either all of the effects of the transaction are in effect after the transaction or none of the effects is in effect. This is by definition only verifiable after a transaction has finished.

Consistency ADS such as secondary indices will be consistent among themselves as well as with the table or other PDS, if any. Such a consistency (compliance to all constraints, if these are declared in the schema, e.g. primary key constraint, foreign key constraints and cardinality constraints) may be verified after the commit or rollback of a transaction. If a single thread of control runs within a transaction, then subsequent operations are expected to see consistent state across all data indices pertaining to a table or similar object. Multiple threads which may share a transaction context are not required to observe a consistent state at all times during the execution of the transaction. Consistency will however always be verifiable after the commit or rollback of any transaction, regardless of the number of threads that have either implicitly or explicitly participated in the transaction. Any intra-transaction parallelism introduced by the SUT will preserve transactional semantics statement-by-statement. If explicit, application created sessions share a transaction context, then this definition of consistency does not hold: for example, if two threads insert into the same table at the same time in the same transaction context, these may or may not see a consistent image of (E)ADS for the parts affected by the other thread. All things will be consistent after the commit or rollback, however, regardless of the number of threads, implicit or explicit that have participated in the transaction.

Isolation Isolation is defined as the set of phenomena that may (or may not) be observed by operations running within a single transaction context. The levels of isolation are defined as follows:

Read uncommitted No guarantees apply.

Read committed A transaction will never read a value that has at no point in time been part of a committed state.

Repeatable read If a transaction reads a value several times during its execution, then it will see the original state with its own modifications so far applied to it. If the transaction itself consists of multiple reading and updating threads then the ambiguities that may arise are beyond the scope of transaction isolation.

Serializable The transactions see values that correspond to a fully serial execution of all client transactions. This is like repeatable read except that if the transaction reads something, and repeats the read, it is

guaranteed that no new values will appear for the same search condition on a subsequent read in the same transaction context. For example, a row that was seen not to exist when first checked will not be seen by a subsequent read. Likewise, counts of items will not be seen to change.

Durability Durability means that once the SUT has confirmed a successful commit, the committed state will survive any instantaneous failure of the SUT (e.g. a power failure, software crash, reboot or the like). Durability is tied to atomicity in that if one part of the changes made by a transaction survives then all parts must survive.

7.3.2 Data Schema

A benchmark may specify restrictions on schema. For example, TPC-H and TPC-DS specify that only certain indices may be declared. In the LDBC context, the matter is more complex since the range of possible SUTs is much broader, including diverse combinations of schema first and schema-less systems and configurations.

7.3.2.1 Schema Declaration

By default, a system may declare no schema at all, as may be the case with RDF or graph DBMSs. If EADSs are declared, then these must be consistently applied to all data within the same workload for a given scale factor. The nature of prohibited EADSs, if any, depends on the benchmark and may be stated in the benchmark specification.

7.3.2.2 Schema-Optional

RDF and graph databases may sometimes be adopted due to their support for schema-last or schema-less operation. It is known that for many cases of RDF with a regular structure, a 1:1 mapping to a relational schema may exist. A benchmark may prohibit the use of such a mapping with the rationale that if the data were purely relational in structure then there would be no point in using RDF or graph DB in the first place. The example of such mapping is Sparqlify (or D2RQ), where SPARQL is directly translated to SQL and run against a relational database.

Use of EADS in a schema-less data model A benchmark may allow use of EADS with a schema-less data model such as RDF with the condition that whilst some data structures may become more efficient, no data structure is prohibited. The schema-less nature may persist but some common structures may benefit from more efficient physical representation.

Benchmarks enforcing schema-first semantics A benchmark may also state that it allows strict schema-first semantics, e.g. SQL, and that the SUT need not make any specific provisions for schema change during the run. For an RDF system this would mean a priori imposing compliance with a data shape or ontology, not with OWL semantics but with semantics close to those of SQL DDL. In such a case, the ontology or data shape may as such be construed to be a valid hint for creation of application specific EADS.

Disclosure of data schema in the FDR In any case, a benchmark must state its policy concerning presence or absence of schema and enforcement thereof. If implementations declare a schema then any schema must be disclosed in full as part of the FDR.

7.3.3 Data Access Transparency

A benchmark may specify that an implementation is not allowed the use of explicit access paths. For example, explicitly specifying which EADS or IADS should be used for any given operation may be prohibited. Furthermore, in scale-out systems, explicit references to data location (other than via values of partitioning keys) may be prohibited. In general, references to internal data representation of an entity, e.g. row in a table, should

be prohibited. Reference should take place via column names in a schema or property URIs in RDF, not via physical offsets or the like.

7.3.4 Query Languages

In general, online transaction processing (OLTP) benchmarks allow implementation via stored procedures, effectively amounting to explicit query plans. Meanwhile, online analytical processing (OLAP) benchmarks prohibit the use of using general-purpose programming languages (e.g. C, C++, Java) for query implementations and only allow domain-specific query languages. Additionally, if a benchmark is analytical and a query is stated using a declarative, explicit control of join order, join type, etc. should be prohibited.

In the graph processing space, there is currently (as of 2021) no standard query language and the systems are more heterogeneous. Therefore, the LDBC situation regarding declarativity is not as simple as that of for example the TPC where queries should be specified in SQL with the additional constraint of omitting any hints for OLAP workloads.

Individual benchmarks are expected to specify their policy concerning using a domain-specific query language or implementing the queries in a general-purpose programming language.

7.3.4.1 Rules for Imperative Implementations Using a General-Purpose Programming Language

An implementation where the queries are written in a general-purpose programming language (including imperative and "API-based" implementations) may choose between semantically equivalent implementations of an operation based on the query parameters. This simulates the behaviour of a query optimizer in the presence of literal values in the query. If an implementation does this, all the code must be disclosed as part of the FDR and the decision must be based on values extracted from the database, not on hard-coded threshold values in the implementation.

The auditor must be able to reliably assess the compliance of an implementation to guidelines specifying these matters. The actual specification remains benchmark-dependent. Borderline cases may be brought to the task force responsible for arbitration.

7.3.4.2 Disclosure of Query Implementations in the FDR

Benchmarks allowing imperative expression of workload should require full disclosure of all query implementation code.

7.3.5 Materialisation

The mix of read and update operations in a workload will determine to which degree precomputation of results is beneficial. The auditor must check that materialised results are kept consistent at the end of each transaction.

7.3.6 Steady State

An online workload must be able to indefinitely keep up the reported throughput. The benchmark definition may put specific restrictions on the duration of individual parts of the workload.

7.3.6.1 Bringing the SUT into Steady State

One implication of this is that an SUT must be able to accommodate inserts at a specific rate for a realistic length of time. For example, if the workload is of an online nature then the SUT should be sized so as not to run out of space for new data for a reasonable duration of time. The TPC-C 180-day rule is an example of this. An analytical benchmark that primarily bulk loads data does not need to reserve as much space for new data. Each benchmark shall state its specific requirements in this respect.

7.3.7 Query Mix

A benchmark consists of multiple different operations that may vary in frequency and duration of individual instances of each operation may vary in function of parameter selection. A benchmark must specify an operation mix and a minimum count of operations that constitutes a compliant benchmark execution.

The auditor must ascertain from the records of a benchmark execution that a sufficient number of operations has indeed taken place for the report. For example, a 1000 GB TPC-H must have at least 7 streams in the throughput test and the workload is to be run twice following bulk load. For LDBC SNB, the run must be at least 2 hours of wall clock, measured time and the count of successful transactions of each type must be in a strictly set ratio with the count of other operations.

Benchmarks shall each specify a minimum count of operations and relative frequencies of operations for a qualifying execution.

7.3.8 System Configuration and System Pricing

A benchmark execution shall produce a full disclosure report which specifies the hardware and software of the SUT, the benchmark implementation version and any specifics that are detailed in the benchmark specification. This clause gives a general minimum for disclosure for the SUT.

7.3.8.1 Details of Machines Driving and Running the Workload

An SUT may consist of one or more pieces of physical hardware. An SUT may include virtual or bare-metal machines in a cloud service. For each distinct configuration, the FDR shall disclose the number of units of the type as well as the following:

1. The used cloud provider (including the region where machines reside, if applicable).
2. Common name of the item, e.g. Dell PowerEdge xxxx or i3.2xlarge instance.
3. Type and number of CPUs, cores & threads per CPU, clock frequency, cache size.
4. Amount of memory, type of memory and memory frequency, e.g. 64GB DDR3 1333MHz.
5. Disk controller or motherboard type if disk controller is on the motherboard.
6. For each distinct type of secondary storage device, the number and specification of the device, e.g. 4xSeagate Constellation 2TB SATA 6Gbit/s.
7. Number and type of network controllers, e.g. 1x Mellanox QDR InfiniBand HCA, PCIE 2.0, 2x1GbE on motherboard. If the benchmark execution is entirely contained on a single machine, it must be stated, and the description of network controllers can be omitted.
8. Number and type of network switches. If multiple switches are used, the wiring between the switches should be disclosed. Only the network switches and interfaces that participate in the run need to be reported. If the benchmark execution is entirely contained on a single machine, it must be stated, and the description of network switches can be omitted.
9. Date of availability of the system as a whole, i.e. the latest date of availability of any part.

The price of the hardware in question must be disclosed. For cloud setups, the price of a dedicated instance for 3 years must be disclosed. The price should reflect the single quantity list price that any buyer could expect when purchasing one system with the given specification. The price may be either an item by item price or a package price if the system is sold as a package. Reported prices should adhere the TPC Pricing Specification 2.5.0 [35, 65].

7.3.8.2 Details of Software Components in the System

The SUT software must be described at least as follows:

1. The units of the SUT software are typically the DBMS and operating system.
2. Name and version of each separately priced piece of the SUT software.

3. If the price of the SUT software is tied to platform or count of concurrent users, these parameters must be disclosed.
4. Price of the SUT software.
5. Date of availability.

Reported prices should adhere the TPC Pricing Specification 2.5.0 [35, 65].

The configuration of the SUT must be reported so as to include the following:

1. The used LDBC specification, driver and data generator version.
2. Complete configuration files of the DBMS, including any general server configuration files, any configuration scripts run on the DBMS for setting up the benchmark run etc.
3. Complete schema of the DBMS, including eventual specification of storage layout.
4. Any OS configuration parameters if other than default, e.g. `vm.swappiness`, `vm.max_map_count` in Linux.
5. Complete source code of any server-side logic, e.g. stored procedures, triggers.
6. Complete source code of driver-side benchmark implementation.
7. Description of system architecture (including software versions, including OS kernel version, DBMS version, Docker version).

7.3.8.3 Audit of System Configuration

The auditor must ascertain that a reported run has indeed taken place on the SUT in the disclosed configuration. The full disclosure shall contain any relevant parameters of the benchmark execution itself, including:

1. Parameters, switches, configuration file for data generation.
2. Complete text of any data loading script or program.
3. Parameters, switches, configuration files for any test driver. If the test driver is not an LDBC supplied open source package or is a modification of such, then the complete text or diff against a specific LDBC package must be disclosed.
4. Test driver output files shall be part of the disclosure. In general, these must at least detail the following:
   i) Time and duration of data load and the timed portion of the benchmark execution.
   ii) Count of each workload item (e.g. query, transaction) successfully executed within the measurement window.
   iii) Min/average/max execution time of each workload item, the specific benchmark shall specify additional details.

Given this information, the number of concurrent database sessions at each point in the execution must be clearly stated. In the case of a cluster database, the possible spreading of connections across multiple server processes must be disclosed.

All parameters included in this section must be reported in the full disclosure report to guarantee that the benchmark run can be reproduced exactly in the future. Similarly, the test sponsor will inform the auditor the scale factor to test. Finally, a clean test system with enough space to store the initial data set, the update streams, substitution parameters and anything that is part of the input and output as well as the benchmark run must be provided.

7.3.9 Benchmark Specifics

Similarly to TPC benchmarks, the LDBC benchmarks prohibit so-called benchmark specials (i.e. extra software modules implemented in the core DBMS logic just to make a selected benchmark run faster are disallowed). Furthermore, upon request of the auditor, the test sponsor must provide all the source code relevant to the benchmark.

7.4 Social Network Benchmark: Interactive Workload

This section specifies a checklist (in the form of individual sections) that a benchmark audit shall cover in case of the SNB Interactive workload. An overview of the benchmark audit workflow is shown in Figure 7.1. The three major phases of the audit are preparing the input data and validation query results (captured by *Preparations* in the figure), validating the correctness of query results returned by the SUT using the validation scale factor and running the benchmark with all the prescribed workloads (*Benchmarking*), and creating the FDR (*Finalisation*). The colour codes capture the responsibilities of performing a step or providing some data in the workflow.

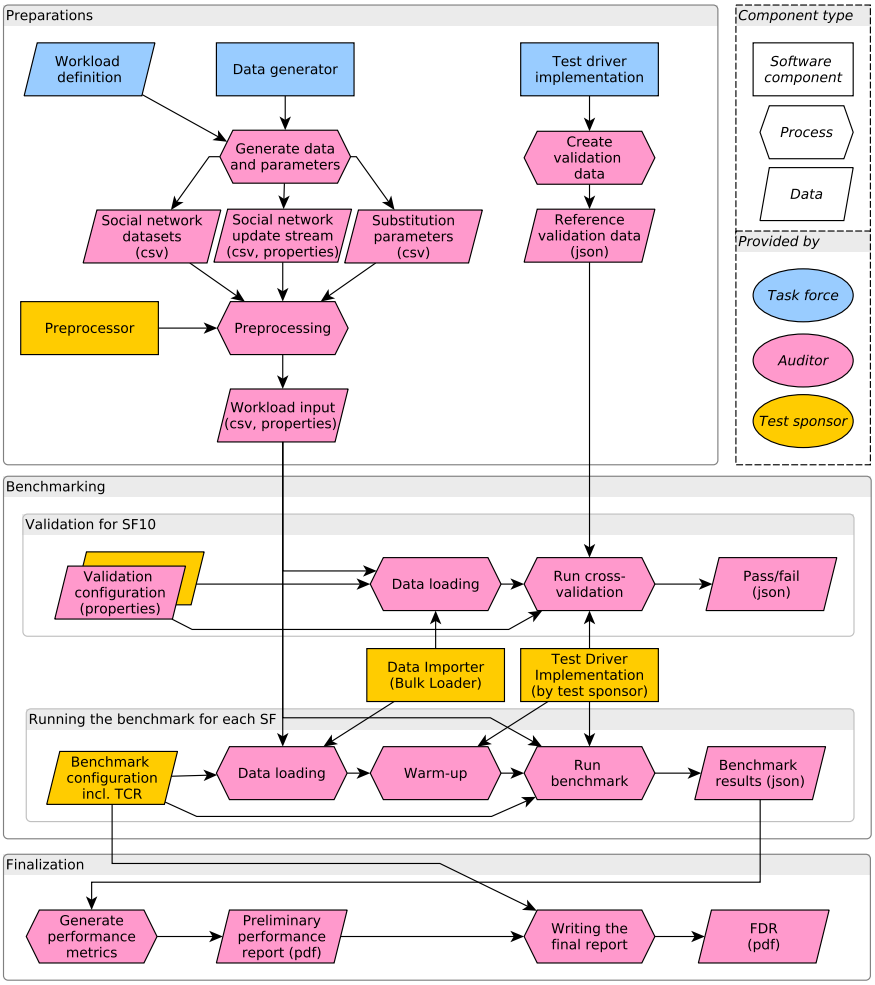


Figure 7.1: Benchmark execution and auditing workflow. For non-audited runs, the implementers perform the steps of the auditor.

A 2015 presentation about the LDBC SNB Benchmark Auditing Process is available in [36].

7.4.1 Scaling

7.4.1.1 Scale Factors

The scale factor of an SNB dataset is the size of the dataset in GiB of CSV (comma-separated values) files. The size of a dataset is characterized by scale factors: SF10, SF30, SF100 etc.. (see Section D.1.1). All datasets contain data for three years of social network activity.

The *validation run* shall be performed on the SF10 dataset (see Section 7.4.6.1). Note that the auditor may perform additional validation runs of the benchmark implementation using smaller datasets (e.g. SF1).

Audited *benchmark runs* shall use SF30 or larger datasets. The rationale behind this decision is to ensure that there is a sufficient number of update operations available to guarantee 2.5 hours of continuous execution (see Section 7.4.7.2).

7.4.1.2 Social Network Datasets

Initial dataset The dataset is divided into a bulk loadable initial database population (90%) and an update stream (10%). These are generated by the SNB data generator. The data generator has options for splitting the dataset into any number of files.

Dependencies between messages in the update stream The update stream contains the latest 10% of the events in the simulated social network. These events form a single serializable sequence in time. Some events will depend on preceding events, for example a message must exist before a reply comment to the message is created. The data generator guarantees that these are separated by at least 10 seconds of simulation time.

Parallel updates The update stream may be broken into arbitrarily many sub-streams. The partition scheme is created by the Datagen. During benchmark execution, the driver preserves dependencies between update operations, such as ensuring not to refer to non-existent entities in updates (e.g. a like is not added to a message which has not been inserted yet).

7.4.2 Data Model and Data Loading

7.4.2.1 Supported Data Models

SNB may be implemented with different data models (e.g. relational, RDF, and different graph data models). The reference schema is provided in the specification using a UML-like notation.

7.4.2.2 Generated Input Data

Storage The data generator produces comma-separated values (CSV) for all data models.

Data format A single attribute has a single data type, as follows:

Identifier This is an integer value foreign key or a URI in RDF. If this is an integer column, the implementation data type should support at least 2^{50} distinct values.

Date A date should support a date range from 0000 to 9999 in the year field.

DateTime A datetime should support a date range from 0000 to 9999 in the year field, with at least millisecond precision.

Short string The string column for names may have a variable length and may have a declared maximum length, e.g. 40 characters.

Long string For example a message content may be a long string that is often short in the data but may not declare a maximum length and must support data sizes of up to 1 MB.

The above is stated in further detail in the benchmark specification, and it shall take precedence over the above in the case of conflict.

A single attribute in the reference schema may not be divided into multiple attributes in the target schema.

Database schema A schema on the DBMS is optional. An RDF implementation for example may work without one. An RDF implementation is allowed to load the RDF reference schema and to take advantage of the data type and cardinality statements therein.

Configuration parameters Datagen configuration parameters, including SF, distributions, number of persons, serialiser (e.g. CsvSingularMergedFK) should be reported.

Primary data structures An RDF, relational, or graph schema may specify system specific options affecting DBMS storage layout. These may for example specify vertical partitioning. Vertical partitioning means anything from a column store layout with per-column allocated storage space to use of explicit column groups. Any mix of row or column-wise storage structures is allowed as long as this is declaratively specified on a per data structure-basis.

Auxiliary data structures Covering indices and clustered indices are allowed. If these are defined, then all replications of data implied by these must be maintained statement by statement, i.e. each auxiliary data structure must be consistent with any other data structures of the table after each data manipulation operation.

A covering index is an index which materialises a specific order of a specific subset or possibly all columns of a table. A clustered index is an index which materialises all columns of a table in a specific order, which order may or may not be that of the primary key of the table. A clustered or covering index may be the primary or only representation of a table.

Any subset of the columns on a covering or clustered index may be used for ordering the data. A hash based index or a combination of a hash based and tree based index are all allowed, in row or column-wise or hybrid forms.

Loading the Data We expect the SUT to provide some means to bulk load the data set either in the form of a dedicated offline loader component or an online loader that allows bulk inserting into a database. The total of the bulk load time and the time for subsequent operations (indexing, computing statistics, etc.) must be reported in the FDR (see Section 7.4.7). As loading can be an expensive operation, it is allowed to conduct the audit such that the loading is only performend once, and the validation/benchmarking phases use the resulting database instance. In practice, this can look like as follows: (1) load the data, (2) compute statistics, uniqueness constraints, keys, indices, etc., (3) shut down the SUT, (4) create a backup of the database (e.g. by copying the directory of the database). For all subsequent runs, the databse shall be restored from the backup.

7.4.3 Precomputation

Precomputation of query results (both interim and end results) is allowed. However, systems must ensure that precomputed results (e.g. materialized views) are kept consistent upon updates.

7.4.4 Benchmark Software Components

LDBC provides a test driver, data generator, and summary reporting scripts. Benchmark implementations shall use a stable version (e.g. 0.3.4) of the test driver.

7.4.4.1 Adaptation of the Test Driver to a DBMS

A qualifying run must use a test driver that adapts the provided test driver to interface with the SUT. Such an implementation, if needed, must be provided by the test sponsor. The parameter generation, result recording, and workload scheduling parts of the test driver should not be changed. The auditor must be given access to the test driver source code used in the reported run.

The test driver produces the following artefacts for each execution as a by product of the run: Start and end timestamps in wall clock time, recorded with microsecond precision. The identifier of the operation and any substitution parameters.

7.4.4.2 Summary of Benchmark Results

A separate test summary tool provided with the test driver analyses the test driver log(s) after a measurement window is completed.

The tool produces for each of the distinct queries and transactions the following summary:

- Run time of query in wall clock time.
- Count of executions.
- Minimum/mean/percentiles/maximum execution time.
- Standard deviation from the average execution time.

The tool produces for the complete run the following summary:

- Operations per second for a given SF (throughput). This is the primary metric of this workload.
- The total execution time in wall clock time.
- The total number of completed operations.

7.4.5 Implementation Language and Data Access Transparency

The queries and updates may be implemented in a domain-specific query language or as procedural code written in a general-purpose programming language (e.g. using the API of the database).

7.4.5.1 Implementations Using a Domain-Specific Query Language

If a domain-specific query language is used, e.g. SPARQL, SQL, Cypher, or Gremlin, then explicit query plans are prohibited in all the read-only queries.[1] The update transactions may still consist of multiple statements, effectively amounting to explicit plans.

Explicit query plans include but are not limited to:

- Directives or hints specifying a join order or join type
- Directives or hints specifying an access path, e.g. which index to use
- Directives or hints specifying an expected cardinality, selectivity, fanout or any other information that pertains to the expected number or results or cost of all or part of the query.

Rationale behind the applied restrictions. The updates are effectively OLTP and, therefore, the customary freedoms apply, including the use of stored procedures, however subject to access transparency. Declarative queries in a benchmark implementation should be such that they could plausibly be written by an application developer. Therefore, their formulation should not contain system specific aspects that an application developer would be unlikely to know. In other words, making a benchmark implementation should not require uncommon sophistication on behalf of the developer. This is regular practice in analytical benchmarks, e.g. TPC-H.

7.4.5.2 Implementations Using a General-Purpose Programming Language

Implementations using a general-purpose programming language for specifying the queries (including procedural, imperative, and API-based implementations) are expected to respect the rules described in Section 7.3.4. For these implementations, the rules in Section 7.4.5.1 do not apply.

7.4.6 Correctness of Benchmark Implementation

7.4.6.1 Validation data set

The scale factor 10 shall be used as validation data set.

7.4.6.2 ACID Compliance

The Interactive workload requires full ACID support from the SUT. See the related section for definitions in Chapter 8.

[1] If the queries are not clearly declarative, the auditor must ensure that they do not specify explicit query plans by investigating their source code and experimenting with the query planner of the system (e.g. using SQL's `EXPLAIN` command).

Expected level of isolation If a transaction reads the database with intent to update, the DBMS must guarantee that repeating the same read within the same transaction will return the same data. This also means that no more and no less data rows must be returned. In other words, this corresponds to snapshot or to serializable isolation. If the database is accessed without transaction context or without intent to update, then the DBMS should provide read committed semantics, e.g. repeating the same read may produce different results but these results may never include effects of pending uncommitted transactions.

Durability and checkpoints A checkpoint is defined as the operation which causes data persisted in a transaction log to become durable outside of the transaction log. Specifically, this means that an SUT restart after instantaneous failure following the completion of the checkpoint may not have recourse to transaction log entries written before the end of the checkpoint.

A checkpoint typically involves a synchronisation barrier at which all data committed prior too the moment is required to be in durable storage that does not depend on the transaction log. Not all DBMSs use a checkpointing mechanism for durability. For example a system may rely on redundant storage of data for durability guarantees against instantaneous failure of a single server.

The measurement window may contain a checkpoint. If the measurement window does not contain one, then the restart test will involve redoing all the updates in the window as part of the recovery test.

The timed window ends with an instantaneous failure of the SUT. Instantaneously killing all the SUT process(es) is adequate for simulating instantaneous failure. All these processes should be killed within one second of each other with an operating system action equivalent to the Unix `kill -9`. If such is not available, then powering down each separate SUT component that has an independent power supply is also possible.

The restart test consists of restarting the SUT process(es) and finishes when the SUT is back online with all its functionality and the last successful update logged by the driver can be seen to be in effect in the database.

If the SUT hardware was powered down, the recovery period does not include the reboot and possible file system check time. The recovery time starts when the DBMS software is restarted.

Recovery The SUT is to be restarted after the measurement window and the auditor will verify that the SUT contains the entirety of the last update recorded by the test driver(s) as successfully committed. The driver or the implementation have to make this information available.

Once an official run has been validated, the recovery capabilities of the system must be tested. The system and the driver must be configured in the same way as in during the benchmark execution. After a warm-up period, an execution of the benchmark will be performed under the same terms as in the previous measured run.

Measuring recovery time At an arbitrary point close to 2 hours of wall clock time during the run, the machine will be shut down. Then, the auditor will restart the database system and will check that the last committed update (in the driver log file) is actually in the database. The auditor will measure the time taken by the system to recover from the failure. Also, all the information about how durability is ensured must be disclosed. If checkpoints are used, these must be performed with a period of 10 minutes at most.

7.4.7 Benchmarking Workflow

A benchmark execution is divided into the following processes (these processes are also shown in Figure 7.1):

Generate data This includes running the data generator, placing the generated files in a staging area, configuring storage, setting up the SUT configuration and preparing any data partitions in the SUT. This may include preallocating database space but may not include loading any data or defining any schema having to do with the benchmark. The `ldbc.snb.interactive.update_interleave` driver parameter must come from the `updateStream.properties` file, which is created by the data generator. That parameter should never be set manually. This parameter signifies the average distance of update operations in the workload.

Preprocessing If needed, the output from the data generator is converted to a format that is loadable by the test-specific implementation of the data importer.

Create validation data Using one of the reference implementations of the benchmark, the reference validation data is obtained in .json format.

Data loading The test sponsor must provide all the necessary documentation and scripts to load the dataset into the database to test. This includes defining the database schema, if any, loading the initial database population, making this durably stored and gathering any optimiser statistics. The system under test must support the different data types needed by the benchmark for each of the attributes at their specified precision. No data can be filtered out, everything must be loaded. The test sponsor must provide a tool to perform arbitrary checks of the data or a shell to issue queries in a declarative language if the system supports it.

Run cross-validation This step uses the data loader to populate the database, but the load is not timed. The validation data set is used to verify the correctness of the SUT. The auditor must load the provided dataset and run the driver in validation mode, which will test that the queries provide the official results. The benchmarking workflow will not go beyond this point unless results match the expected output.

Warm-up Benchmark runs are preceded by a warm-up which must be performed using the LDBC driver.

Run benchmark The bulk load time is reported and is equal to the amount of elapsed wall clock time between starting the schema definition and receiving the confirmation message of the end of statistics gathering. The workflow runs begin after the bulk load is completed. If the run does not directly follow the bulk load, it must start at a point in the update stream that has not previously been played into the database. In other words, a run may only include update events whose timestamp is later than the latest message creation date in the database prior to start of run. The run starts when the first of the test drivers send its first message to the SUT. If the SUT is running in the same process as the driver, the window starts when the driver starts. Also, make sure that the `-rl/--results_log` is enabled. Make sure that all operations are enabled and the frequencies are those for the selected scale factor (see the exact specification of the frequencies in the specification appendix B.1. Scale Factor Statistics for the Interactive Workload).

7.4.7.1 Query Timing During Benchmark Run

A valid benchmark run must last at least 2 hours of wall clock time. In order to be valid, a benchmark run needs to meet the "95% on-time requirement". The `results_log.csv` file contains the actual_start_time and the scheduled_start_time of each of the issued queries. In order to have a valid run, 95% of the queries must meet the following condition:

$$\text{actual\_start\_time} - \text{scheduled\_start\_time} < 1 \text{ second}$$

If the execution of the benchmark is valid, the auditor must retrieve all the files from directory specified by `--results_dir` which includes configuration settings used, results log and results summary. All of which must be disclosed.

7.4.7.2 Measurement Window

Benchmark runs execute the workload on the SUT in two phases (Figure 7.2). First, the SUT must undergo a warm-up period that takes at least 30 minutes. The goal of this is to put the system in a steady state which reflects how it would behave in a normal operating environment. The performance of the operations during warm-up is not considered. Next, the SUT is benchmarked during a two-hour measurement window. Operation times are recorded and checked to ensure the "95% on-time requirement" is satisfied.

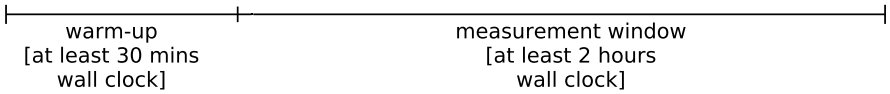


Figure 7.2: Warm-up and measurement window for benchmark run.

The SNB Datagen produces 3 years worth data of which 10% is used for updates (Section 7.4.1.2), i.e. approximately $3 \times 365 \times 0.1 = 109.5$ days $= 2628$ hours. To ensure that the 2.5 hour wall clock period has

enough input data, the lower bound of TCR is defined as 0.001 (if 2628 hours of updates are played back at more than 1000× speed, the benchmark framework runs out of updates to execute). System that can achieve a better compression (i.e. lower TCR value) on a given scale factor should use larger SFs for their benchmark runs – otherwise their total runs will be less than 2.5 hours, making them unsuitable for auditing.

7.4.8 Full Disclosure

Upon successful completion of the audit, an FDR is compiled. In addition to the general requirements, the full disclosure shall cover the following:

- General terms: an executive summary and declaration of the credibility of the audit
- System description and pricing summary: see Section 7.3.8
- Data generation and data loading: see Section 7.4.2.2
- Test driver details: see Section 7.4.4.1
- Performance metrics: see Section 7.4.4.2
- Validation results: see Section 7.4.6.1
- ACID compliance: see Section 7.4.6.2
- List of supplementary materials

To ensure reproducibility of the audited results, a supplementary package is attached to the full disclosure report. This package should contain:

- A README file with instructions specifying how to set up the system and run the benchmark
- Configuration files of the database, including database-level configuration such as buffer size and schema descriptors (if necessary)
- Source code or binary of a generic driver that can be used to interact with the DBMS
- SUT-specific LDBC driver implementation (similarly to the projects in `https://github.com/ldbc/ldbc_snb_interactive`)
- Script or instructions to compile the LDBC Java driver implementation
- LDBC configuration files (`.properties`), including the `time_compression_ratio` values used in the audited runs
- Scripts required to preprocess the input files (if necessary) and to load the data sets into the database
- The implementations of the queries and the update operations, including their complete source code (e.g. declarative queries specifications, stored procedures, etc.)
- Implementation of the ACID test suite
- Binary package of the DBMS (e.g. `.deb` or `.rpm`)

8 ACID Tests

This chapter is based on the TPCTC 2020 conference paper "Towards Testing ACID Compliance in the LDBC Social Network Benchmark" [69], co-authored by several members of the SNB task force.

Verifying ACID compliance is an important step in the benchmarking process for enabling fair comparison between systems. The performance benefits of operating with weaker safety guarantees are well established [28] but this can come at the cost of application correctness. To enable apples vs. apples performance comparisons between systems it is expected they uphold the ACID properties. Currently, LDBC provides no mechanism for validating ACID compliance within the SNB Interactive workflow. A simple solution would be to outsource the responsibility of demonstrating ACID compliance to benchmark implementors. However, the safety properties claimed by a system often do not match observable behaviour [37]. To mitigate this problem, benchmarks such as TPC-C [64] include a number of ACID tests to be executed as part of the benchmarking auditing process. However, we found these tests cannot readily be applied to our context, as they assume lock-based concurrency control and an interactive query API that provides clients with explicit control over a transaction's lifecyle. Modern data systems often use optimistic concurrency control mechanisms [52] and offer a restricted query API, such as only executing transactions as stored procedures [61]. Further, tests that trigger and test row-level locking phenomena, for instance, do not readily map on graph database systems. Lastly, we found these tests are limited in the range of isolation anomalies they cover.

This chapter presents the design of an implementation-agnostic ACID-compliance test suite for the Interactive workload[1]. Our guiding design principle was to be agnostic of system-level implementation details, relying solely on client observations to determine the occurrence of non-transactional behaviour. Thus all systems can be subjected to the same tests and fair comparisons between SNB Interactive performance results can be drawn. Tests are described in the context of a graph database employing the property graph data model [4]. Reference implementations are given in Cypher [26], the *de facto* standard graph query language.

Particular emphasis is given to testing isolation, covering 10 known anomalies including recently discovered anomalies such as *Observed Transaction Vanishes* [11] and *Fractured Reads* [12]. The test suite has been implemented for 5 database systems.[2] A conscious decision was made to keep tests relatively lightweight, as to not add significant overhead to the benchmarking process.

8.1 Background

The tests presented in this chapter are defined on a small core of LDBC SNB schema (extended with properties for versioning) given in Figure 8.1.

[1] We acknowledge verifying ACID-compliance with a finite set of tests is not possible. However, the goal is not an exhaustive quality assurance test of a system's safety properties but rather to demonstrate that ACID guarantees are supported.

[2] Available at `https://github.com/ldbc/ldbc_acid`.

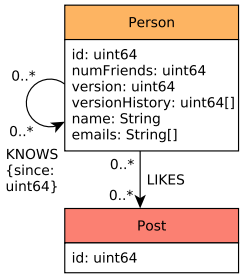


Figure 8.1: Graph schema for the ACID test queries.

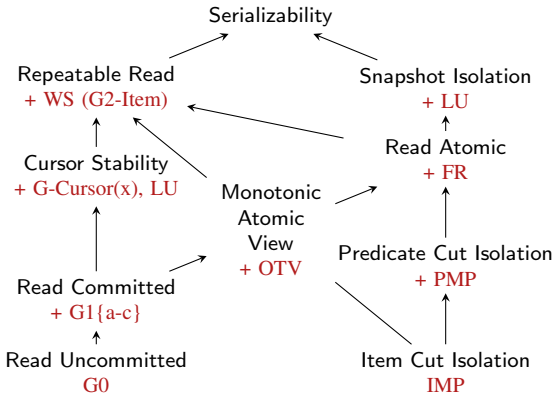


Figure 8.2: Hierarchy of isolation levels as described in [12]. All anomalies are covered except G-Cursor(x).

8.2 Atomicity

Atomicity ensures that either all of a transaction's actions are performed, or none are. Two atomicity tests have been developed. **Atomicity-C** checks for every successful commit message a client receives that any data items inserted or modified are subsequently visible. **Atomicity-RB** checks for every aborted transaction that all its modifications are not visible. Tests are executed as follows: (i) load a graph of `Person` nodes (Listing 8.1) each with a unique `id` and a set of `emails`; (ii) a client executes a full graph scan counting the number of nodes, edges and emails (Listing 8.4) using the result to initialize a counter `committed`; (iii) N transaction instances (Listing 8.2, Listing 8.3) of the required test are then executed, `committed` is incremented for each successful commit; (iii) repeat the full graph scan, storing the result in the variable `finalState`; (iv) perform the anomaly check: `committed=finalState`.

The **Atomicity-C** transaction (Listing 8.2) randomly selects a `Person`, creates a new `Person`, inserts a `KNOWS` edge and appends an `email`. The **Atomicity-RB** transaction (Listing 8.3) randomly selects a `Person`, appends an `email` and attempts to insert a `Person` only if it does not exist. Note, for **Atomicity-RB** if the query API does not offer a `ROLLBACK` statement constraints such as node uniqueness can be utilized to trigger an abort.

```
CREATE (:Person {id: 1, name: 'Alice', emails: ['alice@aol.com']}),
       (:Person {id: 2, name: 'Bob', emails: ['bob@hotmail.com', 'bobby@yahoo.com']})
```

Listing 8.1: Cypher query for creating initial data for the Atomicity transactions.

```
«BEGIN»
MATCH (p1:Person {id: $person1Id})
CREATE (p1)-[k:KNOWS]->(p2:Person)
SET
  p1.emails = p1.emails + [$newEmail],
  p2.id = $person2Id,
  k.creationDate = $creationDate
«COMMIT»
```

Listing 8.2: Atomicity-C Tx.

```
«BEGIN»
MATCH (p1:Person {id: $person1Id})
SET p1.emails = p1.emails + [$newEmail]
«IF» MATCH (p2:Person {id: $person2Id}) exists
«THEN» «ABORT» «ELSE»
CREATE (p2:Person {id: $person2Id, emails: []})
«END»
«COMMIT»
```

Listing 8.3: Atomicity-RB Tx.

```
MATCH (p:Person)
RETURN count(p) AS numPersons, count(p.name) AS numNames, sum(size(p.emails)) AS numEmails
```

Listing 8.4: Atomicity-C/Atomicity-RB: counting entities in the graph.

8.3 Isolation

The gold standard isolation level is Serializability, which offers protection against all possible *anomalies* that can occur from the concurrent execution of transactions. Anomalies are occurrences of non-serializable behaviour. Providing Serializability can be detrimental to performance [28]. Thus systems offer numerous weak isolation levels such as Read Committed and Snapshot Isolation that allow a higher degree of concurrency at the cost of potential non-serializable behaviour. As such, isolation levels are defined in terms of the anomalies they prevent [28, 11]. Figure 8.2 relates isolation levels to the anomalies they proscribe.

SNB Interactive does not require systems to provide Serializability. However, to allow fair comparison systems must disclose the isolation level used during benchmark execution. The purpose of these isolation tests is to verify that the claimed isolation level matches the expected behaviour. To this end, tests have been developed for each anomaly presented in [12]. Formal definitions for each anomaly are reproduced from [1, 12] using their system model which is described below. General design considerations are discussed before each test is described.

8.3.1 System Model

Transactions consist of an ordered sequence of read and write operations to an arbitrary set of data items, book-ended by a `BEGIN` operation and a `COMMIT` or an `ABORT` operation. In a graph database data items are nodes, edges and properties. The set of items a transaction reads from and writes to is termed its *item read set* and *item write set*. Each write creates a *version* of an item, which is assigned a unique timestamp taken from a totally ordered set (e.g. natural numbers) version i of item x is denoted x_i. All data items have an initial *unborn* version $\perp$ produced by an initial transaction $T_\perp$. The unborn version is located at the start of each item's version order. An execution of transactions on a database is represented by a *history*, H, consisting of (i) each transaction's read and write operations, (ii) data item versions read and written and (iii) commit or abort operations.

There are three types of dependencies between transactions, which capture the ways in which transactions can *directly* conflict. *Read dependencies* capture the scenario where a transaction reads another transaction's write. *Antidependencies* capture the scenario where a transaction overwrites the version another transaction reads. *Write dependencies* capture the scenario where a transaction overwrites the version another transaction writes. Their definitions are as follows:

Read-Depends Transaction T_j *directly read-depends* (wr) on T_i if T_i writes some version x_k and T_j reads x_k.

Anti-Depends Transaction T_j *directly anti-depends* (rw) on T_i if T_i reads some version x_k and T_j writes x's next version after x_k in the version order.

Write-Depends Transaction T_j *directly write-depends* (ww) on T_i if T_i writes some version x_k and T_j writes x's next version after x_k in the version order.

Using these definitions, from a history H a *direct serialization graph DSG*(H) is constructed. Each node in the *DSG* corresponds to a committed transaction and edges correspond to the types of direct conflicts between transactions. Anomalies can then be defined by stating properties about the *DSG*.

The above *item-based* model can be extended to handle *predicate-based* operations [1]. Database operations are frequently performed on set of items provided a certain condition called the *predicate*, P holds. When a transaction executes a read or write based on a predicate P, the database selects a version for each item to which P applies, this is called the version set of the predicate-based denoted as *Vset*(P). A transaction T_j changes the matches of a predicate-based read $r_i(P_i)$ if T_i overwrites a version in *Vset*(P_i).

8.3.2 General Design

Isolation tests begin by loading a *test graph* into the database. Configurable numbers of *write clients* and *read clients* then execute a sequence of transactions on the database for some configurable time period. After execution, results from read clients are collected and an *anomaly check* is performed. In some tests an additional full graph scan is performed after the execution period in order to collect information required for the anomaly check.

The guiding principle behind test design was the preservation of data item's version history – the key ingredient needed in the system model formalization which is often not readily available to clients, if preserved at all. Several anomalies are closely related, tests therefore had to be constructed such that other anomalies could not interfere with or mask the detection of the targeted anomaly. Test descriptions provide (i) informal and formal anomaly definitions, (ii) the required test graph, (iii) description of transaction profiles write and read clients execute, and (iv) reasoning for why the test works.

8.3.3 Dirty Write

Informally, a *Dirty Write* (Adya's G0 [1]) occurs when updates by conflicting transactions are interleaved. For example, say T_i and T_j both modify items $\{x, y\}$. If version x_i precedes version x_j and y_j precedes version y_i a G0 anomaly has occurred. Preventing G0 is especially important in a graph database in to order to maintain *Reciprocal Consistency* [67].

Definition. A history H exhibits phenomenon G0 if $DSG(H)$ contains a directed cycle consisting entirely of write-dependency edges.

Test. Load a test graph containing pairs of `Person` nodes connected by a `KNOWS` edge. Assign each `Person` a unique `id` and each `Person` and `KNOWS` edge a `versionHistory` property of type list (initially empty). During the execution period, write clients execute a sequence of G0 T_{W} instances, Listing 8.5. This transaction appends its ID to the `versionHistory` property for each entity in the `Person` pair it matches. Note, transaction IDs are assumed to be globally unique. After execution, a read client issues a G0 T_{R} for each `Person` pair in the graph, Listing 8.6. Retrieving the `versionHistory` for each entity (2 `Persons` and 1 `KNOWS` edge) in a `Person` pair.

Anomaly check. For each `Person` pair in the test graph: (i) prune each `versionHistory` list to remove any version numbers that do not appear in all lists; needed to account for interference from *Lost Update* anomalies (Section 8.3.8), (ii) perform an element-wise comparison between `versionHistory` lists for each entity, (iii) if lists do not agree a G0 anomaly has occurred.

Why it works. Each G0 T_{W} effectively creates a new version of a `Person` pair. Appending the transaction ID preserves the version history of each entity in the `Person` pair. In a system that prevents G0, each entity of the `Person` pair should experience the *same* updates, in the *same* order. Hence, each position in the `versionHistory` lists should be equivalent. The additional pruning step is needed as *Lost Updates* overwrite a version, effectively erasing it from the history of a data item.

```
MATCH
  (p1:Person {id: $person1Id})
  -[k:KNOWS]->(p2:Person {id: $person2Id})
SET p1.versionHistory = p1.versionHistory + [$tId]
SET p2.versionHistory = p2.versionHistory + [$tId]
SET k.versionHistory  = k.versionHistory  + [$tId]
```

Listing 8.5: Dirty Write (G0) T_{W}.

```
MATCH (p1:Person {id: $person1Id})
-[k:KNOWS]->(p2:Person {id: $person2Id})
RETURN
  p1.versionHistory AS p1VersionHistory,
  k.versionHistory  AS kVersionHistory,
  p2.versionHistory AS p2VersionHistory
```

Listing 8.6: Dirty Write (G0) T_{R}.

8.3.4 Dirty Reads

Aborted Reads

Informally, an *Aborted Read* (G1a) anomaly occurs when a transaction reads the updates of a transaction that later aborts.

Definition. A history H exhibits phenomenon G1a if H contains an aborted transaction T_i and a committed transaction T_j such that T_j reads a version written by T_i.

Test. Load a test graph containing only `Person` nodes into the database. Assign each `Person` a unique `id` and `version` initialized to 1; any odd number will suffice. During execution, write clients execute a sequence of G1a T_W instances, Listing 8.7. Selecting a random `Person` `id` to populate each instance. This transaction attempts to set `version=2` (any even number will suffice) but always aborts. Concurrently, read clients execute a sequence of G1a T_R instances, Listing 8.8. This transaction retrieves the `version` property of a `Person`. Read clients store results, which are pooled after execution has finished.

Anomaly check. Each read should return `version=1` (or any odd number). Otherwise, a G1a anomaly has occurred.

Why it works. Each transaction that attempts to set `version` to an even number *always* aborts. Therefore, if a transaction reads `version` to be an even number, it must have read the write of an aborted transaction.

```
MATCH (p:Person {id: $personId})
SET p.version = 2
«SLEEP($sleepTime)»
«ABORT»
```

Listing 8.7: Aborted Read (G1a) T_W.

```
MATCH (p:Person {id: $personId})
RETURN p.version
```

Listing 8.8: Aborted Read (G1a) T_R.

```
MATCH (p:Person {id: $personId})
SET p.version = $even
«SLEEP($sleepTime)»
SET p.version = $odd
```

Listing 8.9: Interm. Read (G1b) T_W.

```
MATCH (p:Person {id: $personId})
RETURN p.version
```

Listing 8.10: Interm. Read (G1b) T_R.

Intermediate Reads

Informally, an *Intermediate Read* (Adya's G1b [1]) anomaly occurs when a transaction reads the intermediate modifications of other transactions.

Definition. A history H exhibits phenomenon G1b if H contains a committed transaction T_i that reads a version of an object x_m written by transaction T_j, and T_j also wrote a version x_n such that $m < n$ in x's version order.

Test. Load a test graph containing only `Person` nodes into the database. Assign each `Person` a unique `id` and `version` initialized to 1; any odd number will suffice. During execution, write clients execute a sequence of G1b T_W instances, Listing 8.9. This transaction sets `version` to an even number, then an odd number before committing. Concurrently read-clients execute a sequence of G1b T_R instances, Listing 8.10. Selecting a `Person` by `id` and retrieving its `version` property. Read clients store results which are collected after execution has finished.

Anomaly check. Each read of `version` should be an odd number. Otherwise, a G1b anomaly has occurred.

Why it works. The final version installed by an G1b T_W instance is *never* an even number. Therefore, if a transaction reads `version` to be an even number it must have read an intermediate version.

```
MATCH (p1:Person {id: $person1Id}) SET p1.version = $transactionId
MATCH (p2:Person {id: $person2Id}) RETURN p2.version
```

Listing 8.11: G1c T_{RW}.

Circular Information Flow

Informally, a *Circular Information Flow* (Adya's G1c [1]) anomaly occurs when two transactions affect each other; i.e. both transactions write information the other reads. For example, transaction T_i reads a write by transaction T_j and transaction T_j reads a write by T_i.

Definition. A history H exhibits phenomenon G1c if $DSG(H)$ contains a directed cycle that consists entirely of read-dependency and write-dependency edges.

Test. Load a test graph containing only `Person` nodes into the database. Assign each `Person` a unique `id` and `version` initialized to 0. Read-write clients are required for this test, executing a sequence of G1c T_{RW}, Listing 8.11. This transaction selects two different `Person` nodes, setting the `version` of one `Person` to the transaction ID and retrieving the `version` from the other. Note, transaction IDs are assumed to be globally unique. Transaction results are stored in format `(txn.id, versionRead)` and collected after execution.

Anomaly check. For each result, check the result of the transaction the `versionRead` corresponds to, did not read the transaction of that result. If so a G1c anomaly has occurred.

Why it works. Consider the result set: $\{(T_1, T_2), (T_2, T_3), (T_3, T_2)\}$. T_1 reads the version written by T_2 and T_2 reads the version written by T_3. Here information flow is unidirectional from T_1 to T_2. However, T_2 reads the version written by T_3 and T_2 reads the version written by T_3. Here information flow is circular from T_2 to T_3 and T_3 to T_2. Thus a G1c anomaly has been detected.

8.3.5 Cut Anomalies

Item-Many-Preceders

Informally, an *Item-Many-Preceders* (IMP) anomaly [11] occurs if a transaction observes multiple versions of the same item (e.g. transaction T_i reads versions x_1 and x_2). In a graph database this can be multiple reads of a node, edge, property or label. Local transactions (involving a single data item) occur frequently in graph databases, e.g. in *"Retrieve content of a message"* `IS 4` .

Definition. A history H exhibits IMP if $DSG(H)$ contains a transaction T_i such that T_i directly *item-read-depends* on x by more than one other transaction.

Test. Load a test graph containing `Person` nodes. Assign each `Person` a unique `id` and `version` initialized to 1. During execution write clients execute a sequence of IMP T_{W} instances, Listing 8.12. Selecting a random `id` and installing a new version of the `Person`. Concurrently read clients execute a sequence of IMP T_{R} instances, Listing 8.13. Performing multiple reads of the same `Person`; successive reads can be separated by some artificially injected wait time to make conditions more favourable for detecting an anomaly. Both reads within an IMP T_{R} transaction are returned, stored and collected after execution.

Anomaly check. Each IMP T_{R} result set `(firstRead, secondRead)` should contain the *same* `Person` version. Otherwise, an IMP anomaly has occurred.

Why it works. By performing successive reads within the same transaction this test checks that a system ensures consistent reads of the same data item. If the version changes then a concurrent transaction has modified the data item and the reading transaction is not protected from this change.

```
MATCH (p:Person {id: $personId})
SET p.version = p.version + 1
```

Listing 8.12: IMP T_{W}.

```
MATCH (p1:Person {id: $personId})
WITH p1.version AS firstRead
«SLEEP($sleepTime)»
MATCH (p2:Person {id: $personId})
RETURN firstRead,
  p2.version AS secondRead
```

Listing 8.13: IMP T_{R}.

```
MATCH (pe:Person {id: $personId}), (po:Post {id: $postId)
CREATE (pe)-[:LIKES]->(po)
```

Listing 8.14: PMP T_{W}.

```
MATCH (po1:Post {id: $postId})<-[:LIKES]-(pe1:Person)
WITH count(pe1) AS firstRead
«SLEEP($sleepTime)»
MATCH (po2:Post {id: $postId})<-[:LIKES]-(pe2:Person)
RETURN firstRead,
  count(pe2) AS secondRead
```

Listing 8.15: PMP T_{R}.

Predicate-Many-Preceders

Informally, a *Predicate-Many-Preceders* (PMP) anomaly [11] occurs if a transaction observes different versions resulting from the same predicate read (e.g. T_i reads $Vset(P_i) = \{x_1\}$ and $Vset(P_i) = \{x_1, y_2\}$). Pattern matching is a common predicate read operation in a graph database, e.g. query *"Find friends and friends of friends that have been to given countries"* `IC 3` .

Definition. A history H exhibits the phenomenon PMP if, for all predicate-based reads $r_i(P_i : Vset(P_i))$ and $r_j(P_j : Vset(P_j))$ in T_k such that the logical ranges of P_i and P_j overlap (call it P_o), the set of transactions that change the matches of P_o for r_i and r_j differ.

Test. Load a test graph containing `Person` and `Post` nodes. Within each node type assign unique IDs. During execution write clients execute a sequence of PMP T_{W} instances, inserting a `LIKES` edge between a randomly selected `Person` and `Post`, shown in Listing 8.14. Concurrently read clients execute a sequence of PMP T_{R} instances, Listing 8.15. Performing multiple reads of the pattern `(po:Post)<-[:LIKES]-(p:Person)` and counting the number of `LIKES` edges; successive reads can be separated by some artificially injected wait time to make conditions more favourable for detecting an anomaly. Both predicate reads within a PMP T_{R} transaction are returned, stored and collected after test execution.

Anomaly check. For each PMP T_{R} transaction result set `(firstRead, secondRead)`, the `firstRead` should be equal to `secondRead`. Otherwise, a PMP anomaly has occurred.

Why it works. By performing successive predicate reads and counting the number of `LIKES` edges within the same transaction this test checks that a system ensures consistent reads of the same predicate. If the number of `LIKES` edges changes then a concurrent transaction has inserted a new `LIKES` edge and the reading transaction is not protected from this change.

8.3.6 Observed Transaction Vanishes

Informally, an *Observed Transaction Vanishes* (OTV) anomaly [11] occurs when a transaction observes part of another transaction's updates but not all of them (e.g. T_1 writes x_1 and y_1 and T_2 reads x_1 and $y_\perp$). Before

formally defining OTV the *Unfolded Serialization Graph (USG)* must be introduced [1]. The *USG* is specified for an individual transaction, T_i and a history, H and is denoted by $USG(H, T_i)$. In a *USG* the T_i node is split into multiple nodes, one for each action read $r_i(\cdot)$ or write $w_i(\cdot)$ within the transaction. The dependency edges are now incident on the relevant event of T_i. Additionally, actions within T_i are connected by an *order edge* e.g. if T_i reads object y_j then immediately writes on object x an order edge exists from $w_i(x_i)$ to $r_i(y_j)$.

Definition. A history H exhibits phenomenon OTV if $USG(H, T_i)$ contains a directed cycle consisting of (i) exactly one read dependency edge induced by data item x from T_j to T_i and (ii) a set of edges induced by data item y containing at least one anti dependency edge from T_i to T_j. Additionally, T_i's read from y precedes its read from x.

Test. Load a test graph containing a set of cycles of length 4 of `Persons` with same `name` connected by `Knows` edges. Assign each `Person` an `id`, `name` and `version` property (initialized to 1). Note, `id` must be unique across nodes and `name` must be unique across cycles. During execution write clients select a `name`, `id` and executes a sequence of OTV T_{W} instances, Listing 8.16. This transaction effectively creates a new version of a given cycle. Concurrently read-clients execute a sequence of OTV T_{R} instances, Listing 8.17. Matching a given cycle and performing multiple reads. Both reads within an OTV T_{R} are returned, stored and collected after execution.

Anomaly check. For each OTV T_{R} result set `(firstRead,secondRead)`, the maximum `version` in the `firstRead` should be less than or equal to the minimum `version` in the `secondRead`. Otherwise, an OTV anomaly has occurred.

Why it works. OTV T_{W} installs a new version of a cycle by updating the `version` property of each `Person`. Therefore when matching a cycle once a transaction has observed some `version` it should *at least* observe this version for every remaining entity in the cycle. Unfortunately, this cannot be deduced from a single read of the cycle as results from matching cycles often does not preserve the order in which graph entities were read. This is solved by making multiple reads of the cycle. The maximum `version` of the `firstRead` determines the minimum `version` of `secondRead`. If this condition is violated then a transaction has observed the effects of a transaction in the `firstRead` then subsequently failed to observe it in the `secondRead` – the observed transaction has vanished!

```
MATCH path =
  (n:Person {id: $personId})
  -[:KNOWS*..4]->(n)
UNWIND nodes(path)[0..4] AS p
SET p.version = p.version + 1
```

Listing 8.16: OTV/FR T_{W}.

```
MATCH p1=(n1:Person {id: $personId})-[:KNOWS*..4]->(n1)
RETURN extract(p IN nodes(p1) | p.version) AS firstRead
«SLEEP($sleepTime)»
MATCH p2=(n2:Person {id: $personId})-[:KNOWS*..4]->(n2)
RETURN extract(p IN nodes(p2) | p.version) AS secondRead
```

Listing 8.17: OTV/FR T_{R}.

8.3.7 Fractured Read

Informally, a *Fractured Read* (FR) anomaly [12] occurs when a transaction reads *across* transaction boundaries. For example, if T_1 writes x_1 and y_1 and T_3 writes x_3. If T_2 reads x_1 and y_1, then repeats its read of x and reads x_3 a fractured read has occurred.

Definition. A transaction T_j exhibits phenomenon FR if transaction T_i writes versions x_a and y_b (in any order, where x and y may or may not be distinct items), T_j reads version x_a and version y_c, and $c < b$.

Test. Same as the OTV test.

Anomaly check. For each FR T_{R} (Listing 8.17) result set `(firstRead, secondRead)`, all `versions` across both version sets should be equal. Otherwise, an FR anomaly has occurred.

Why it works. FR T_{W} installs a new version of a cycle by updating the `version` properties on each `Person`. When FR T_{R} observes a `version` every subsequent read in that cycle should read the *same* `version` as FR T_{W} (Listing 8.16) installs the same `version` for all `Person` nodes in the cycle. Thus, if it observes a different `version` it has observed the effect of a different transaction and has read across transaction boundaries.

8.3.8 Lost Update

Informally, a *Lost Update* (LU) anomaly [12] occurs when two transactions concurrently attempt to make conditional modifications to the same data item(s).

Definition. A history H exhibits phenomenon LU if $DSG(H)$ contains a directed cycle having one or more antidependency edges and all edges are induced by the same data item x.

Test. Load a test graph containing `Person` nodes. Assign each `Person` a unique `id` and a property `numFriends` (initialized to 0). During execution write clients execute a sequence of LU T_{W} instances, Listing 8.18. Choosing a random `Person` and incrementing its `numFriends` property. Clients store local counters (`expNumFriends`) for each `Person`, which is incremented each time a `Person` is selected *and* the LU T_{W} instance successfully commits. After the execution period the `numFriends` is retrieved for each `Person` using LU T_{R} in Listing 8.19 and `expNumFriends` are pooled from write clients for each `Person`.

Anomaly check. For each `Person` its `numFriends` property should be equal to the (global) `expNumFriends` for that `Person`.

Why it works. Clients know how many successful LU T_{W} instances were issued for a given `Person`. The observable `numFriends` should reflect this ground truth, otherwise, an LU anomaly must have occurred.

```
MATCH (p:Person {id: $personId})
SET p.numFriends = p.numFriends + 1
```

Listing 8.18: Lost Update T_{W}.

```
MATCH (p:Person {id: $personId})
RETURN p.numFriends AS numFriends
```

Listing 8.19: Lost Update T_{R}.

8.3.9 Write Skew

Informally, *Write Skew* (WS) occurs when two transactions simultaneously attempted to make *disjoint* conditional modifications to the same data item(s). It is referred to as G2-Item in [1, 25].

Definition. A history H exhibits WS if $DSG(H)$ contains a directed cycle having one or more antidependency edges.

Test. Load a test graph containing n pairs of `Person` nodes `(p1, p2)` for $k = 0, \ldots, n-1$, where the kth pair gets IDs `p1.id = 2*k+1` and `p2.id = 2*k+2`, and values `p1.value = 70` and `p2.value = 80`. There is a constraint: `p1.value + p2.value > 0`. During execution write clients execute a sequence of WS T_{W} instances, Listing 8.20. Selecting a random `Person` pair and decrementing the `value` property of one `Person` provided doing so would not violate the constraint. After execution the database is scanned using WS T_{R}, Listing 8.21.

Anomaly check. For each `Person` pair the constraint should hold true, otherwise, a WS anomaly has occurred.

Why it works. Under no Serializable execution of WS T_{W} instances would the constraint `p1.value + p2.value > 0` be violated. Therefore, if WS T_{R} returns a violation of this constraint it is clear a WS anomaly has occurred.

```
MATCH (p1:Person {id: $person1Id}),
      (p2:Person {id: $person2Id})
«IF (p1.value+p2.value < 100)» «THEN» «ABORT» «END»
«SLEEP($sleepTime)»
pId = «pick randomly between personId1, personId2»
MATCH (p:Person {id: $pId})
SET p.value = p.value - 100
«COMMIT»
```

Listing 8.20: WS T_{W}.

```
MATCH (p1:Person),
      (p2:Person {id: p1.id+1})
WHERE p1.value + p2.value <= 0
RETURN
  p1.id AS p1id,
  p1.value AS p1value,
  p2.id AS p2id,
  p2.value AS p2value
```

Listing 8.21: WS T_{R}.

8.4 Consistency and Durability Tests

While this chapter mainly focused on *atomicity* and *isolation* from the ACID properties, we provide a short overview of the other two aspects.

Durability is a hard requirement for SNB Interactive and checking it is part of the auditing process. The durability test requires the execution of the SNB Interactive workload and uses the LDBC workload driver. Note, the database and the driver must be configured in the same way as would be used in the performance run. First, the database is subject to a warm-up period. Then after 2 hours of simulation execution, the database processes will be terminated, possibly by disconnecting the entire machine or by a hard process kill. Note that turning the machine off may not be possible in cloud tests. The database system is then restarted and each client issues a read for the value of the last entity (node or edge) it received a successful commit message for, that should produce a correct response.

Consistency is defined in terms of constraints: the database remains consistent under updates; i.e. no constraint is violated. Relational database systems usually support primary- and foreign-key constraints, as well as domain constraints on column values and sometimes also support simple within-row constraints. Graph database systems have a diversity of interfaces and generally do not support constraints, beyond sometimes domain and primary key constraints (in case indices are supported). As such, we leave them out of scope for LDBC SNB. However, we do note that we anticipate that graph database systems will evolve to support constraints in the future. Beyond equivalents of the relational ones, property graph systems might introduce graph-specific constraints, such as (partial) compliance to a schema formulated on top of property graphs, rules that guide the presence of labels or structural graph constraints such as connectedness of the graph, absence of cycles, or arbitrary well-formedness constraints [59].

9 Related Work

A detailed list of LDBC publications is curated at `http://ldbcouncil.org/publications`.

9.1 ACID Tests in Other Benchmarks

The challenge of verifying ACID-compliance has been addressed before by transactional benchmarks. For example, TPC-C [64] provides a suite of ACID tests. However, the isolation tests are reliant on lock-based concurrency control, hence are not generalizable across systems. Also, the transactional anomaly test coverage is limited to only four anomalies. The authors of [20] augment the popular YCSB framework for benchmarking transactional NewSQL systems, including a *validation phase* that detects and quantifies consistency anomalies. They permit the definition of arbitrary integrity constraints, checking they hold before and after a benchmark run. Such an approach is not possible within SNB Interactive due to the restrictive nature of transactional updates and the distinct lack of application-level constraints.

The Hermitage project [39] with the goal of improving understanding of weak isolation, developed a range of hand-crafted isolation tests. This test suite has much higher anomaly coverage but suffers from a problem similar to TPC-C. Test execution is performed by hand, opening multiple terminals to step through the tests.[1] The Jepsen project [37] is not a benchmark rather it addresses correctness testing, traditionally focusing on distributed systems under various failure modes. Most of Jepsen's transactional tests adopt a similar approach to us, executing a suite of transactions with hand-proven invariants. However recently, the project has spawned Elle [38] a black-box transactional anomaly checker. Elle does not rely on hand-crafted tests and can detect every anomaly in [1] (except for predicate-based anomalies) from an arbitrary transaction history.

9.2 Graph Processing Benchmarks

Recent graph benchmarking initiatives focus on three key areas:

1. transactional workloads consisting of interactive read and update queries (OLTP) aiming at graph databases that explore small portions of the graph in each query [13, 8, 19, 24, 42],
2. graph analysis algorithms (e.g. PageRank) computed in bulk, typically expressed in cluster frameworks with graph APIs, rather than high-level queries [10, 22, 49, 33],
3. pattern matching and inferencing on semantic data [30, 58, 47, 3, 63].

The SIGMOD 2014 Programming Contest defined queries on the Social Network Benchmark schema with a mix of subgraph projection and graph analytics [21].

The challenges of using benchmarks correctly are described in [55].

The Interactive queries were used in paper [51] to compare the performance of Gremlin, Cypher, SQL and SPARQL query engines.

9.3 Scalable Graph Generators

A recent survey [17] studied 38 graph generators, finding that only 4 of them supported generating updates and, intriguingly, even these generators only yield insertions and simple deletions at best. *LinkBench* [8] defines primitive delete operations targeting a single node or a single edge. *XGDBench* [19] defines an operation that deletes a single node. The *Social Network Intelligence BenchMark* (SIB) [16] (a precursor to LDBC SNB) requires the deletion of individual nodes (posts/photos).

[1] We initially experimented with Hermitage but found it difficult to induce anomalies that relied on fast timings due to some graph databases offering limited client-side control over transactions, with all statements submitted in one batch.

Bibliography

[1] Atul Adya. "Weak consistency: A generalized theory and optimistic implementations for distributed transactions". Ph.D. dissertation. MIT, 1999.

[2] Hazim Almuhimedi et al. "Tweets are forever: a large-scale quantitative analysis of deleted tweets". In: *Computer Supported Cooperative Work, CSCW 2013, San Antonio, TX, USA, February 23-27, 2013*. Ed. by Amy S. Bruckman et al. ACM, 2013, pp. 897–908. DOI: 10.1145/2441776.2441878.

[3] Günes Aluç et al. "Diversified Stress Testing of RDF Data Management Systems". In: *ISWC*. 2014, pp. 197–212. DOI: 10.1007/978-3-319-11964-9_13.

[4] Renzo Angles et al. "Foundations of Modern Query Languages for Graph Databases". In: *ACM Comput. Surv.* 50.5 (2017), 68:1–68:40. DOI: 10.1145/3104031.

[5] Renzo Angles et al. "G-CORE: A Core for Future Graph Query Languages". In: *SIGMOD*. ACM, 2018, pp. 1421–1432. DOI: 10.1145/3183713.3190654.

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

[7] Renzo Angles et al. "The Linked Data Benchmark Council: A graph and RDF industry benchmarking effort". In: *SIGMOD Record* 43.1 (2014), pp. 27–31. DOI: 10.1145/2627692.2627697.

[8] Timothy G. Armstrong et al. "LinkBench: A database benchmark based on the Facebook social graph". In: *SIGMOD*. 2013, pp. 1185–1196. DOI: 10.1145/2463676.2465296.

[9] Alex Averbuch and Arnau Prat-Pérez. *Benchmark design for navigational pattern matching benchmarking*. Tech. rep. http://ldbcouncil.org/sites/default/files/LDBC_D3.3.34.pdf. Linked Data Benchmark Council, 2014.

[10] David A. Bader and Kamesh Madduri. "Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors". In: *HiPC*. 2005, pp. 465–476. DOI: 10.1007/11602569_48.

[11] Peter Bailis et al. "Highly Available Transactions: Virtues and Limitations". In: *VLDB* (2013). DOI: 10.14778/2732232.2732237.

[12] Peter Bailis et al. "Scalable Atomic Visibility with RAMP Transactions". In: *ACM Trans. Database Syst.* (2016). DOI: 10.1145/2909870.

[13] Sumita Barahmand and Shahram Ghandeharizadeh. "BG: A Benchmark to Evaluate Interactive Social Networking Actions". In: *CIDR*. 2013. URL: http://cidrdb.org/cidr2013/Papers/CIDR13_Paper93.pdf.

[14] Mauro Barone and Michele Coscia. "Birds of a feather scam together: Trustworthiness homophily in a business network". In: *Social Networks* 54 (2018), pp. 228–237. DOI: 10.1016/j.socnet.2018.01.009.

[15] Maciej Besta et al. "Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries". In: *CoRR* abs/1910.09017 (2019). URL: http://arxiv.org/abs/1910.09017.

[16] Peter Boncz et al. *Social Network Intelligence BenchMark*. 2013. URL: https://www.w3.org/wiki/Social_Network_Intelligence_BenchMark.

[17] Angela Bonifati et al. "Graph Generators: State of the Art and Open Challenges". In: *ACM Comput. Surv.* 53.2 (2020), 36:1–36:30. DOI: 10.1145/3379445.

[18] Federico Busato et al. "Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs". In: *HPEC*. IEEE, 2018, pp. 1–7. DOI: 10.1109/HPEC.2018.8547541.

[19] Miyuru Dayarathna and Toyotaro Suzumura. "Graph database benchmarking on cloud environments with XGDBench". In: *Autom. Softw. Eng.* 21.4 (2014), pp. 509–533. DOI: 10.1007/s10515-013-0138-7.

[20] Akon Dey et al. "YCSB+T: Benchmarking web-scale transactional databases". In: *ICDE*. IEEE Computer Society, 2014, pp. 223–230. DOI: 10.1109/ICDEW.2014.6818330.

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

[22] Benedikt Elser and Alberto Montresor. "An evaluation study of BigData frameworks for graph processing". In: *Big Data*. 2013, pp. 60–67. DOI: 10.1109/BigData.2013.6691555.

[23] Orri Erling, Alex Averbuch, and Peter Boncz. *LDBC Benchmark Auditing Policies*. Tech. rep. http://ldbcouncil.org/sites/default/files/LDBC_D6.6.3.pdf. Linked Data Benchmark Council, 2014.

[24] Orri Erling et al. "The LDBC Social Network Benchmark: Interactive Workload". In: *SIGMOD*. 2015, pp. 619–630. DOI: 10.1145/2723372.2742786.

[25] Alan Fekete et al. "Making snapshot isolation serializable". In: *ACM Trans. Database Syst.* 30.2 (2005), pp. 492–528. DOI: 10.1145/1071610.1071615.

[26] Nadime Francis et al. "Cypher: An Evolving Query Language for Property Graphs". In: *SIGMOD*. ACM, 2018, pp. 1433–1445. DOI: 10.1145/3183713.3190657.

[27] Goetz Graefe. "Query Evaluation Techniques for Large Databases". In: *ACM Comput. Surv.* 25.2 (1993), pp. 73–170. DOI: 10.1145/152610.152611.

[28] Jim Gray et al. "Granularity of Locks and Degrees of Consistency in a Shared Data Base". In: *IFIP Working Conference on Modelling in Data Base Management Systems*. 1976, pp. 365–394.

[29] Alastair Green et al. "Updating Graph Databases with Cypher". In: *PVLDB* 12.12 (2019), pp. 2242–2253. DOI: 10.14778/3352063.3352139.

[30] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. "LUBM: A benchmark for OWL knowledge base systems". In: *J. Web Sem.* 3.2-3 (2005), pp. 158–182. DOI: 10.1016/j.websem.2005.06.005.

[31] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. "Graph Grammars with Negative Application Conditions". In: *Fundam. Inform.* 26.3/4 (1996), pp. 287–313. DOI: 10.3233/FI-1996-263404.

[32] Torsten Hoefler and Roberto Belli. "Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results". In: *SC*. ACM, 2015, 73:1–73:12. DOI: 10.1145/2807591.2807644.

[33] Alexandru Iosup et al. "LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms". In: *VLDB* 9.13 (2016), pp. 1317–1328. DOI: 10.14778/3007263.3007270.

[34] Alexandru Iosup et al. "The LDBC Graphalytics Benchmark". In: *CoRR* abs/2011.15028 (2020). URL: https://arxiv.org/abs/2011.15028.

[35] Moritz Kaufmann. *Examining the TPC Pricing Specification 2.0.0*. Presented at the 9th LDBC TUC. 2017. URL: http://wiki.ldbcouncil.org/pages/viewpage.action?pageId=59277315&preview=/59277315/75431947/LDBCPricing2.pdf.

[36] Moritz Kaufmann. *LDBC SNB Benchmark Auditing*. Presented at the 6th LDBC TUC. 2015. URL: https://www.slideshare.net/ldbcouncil/ldbc-snb-benchmark-auditing.

[37] Kyle Kingsbury. *Jepsen Analyses*. http://jepsen.io/analyses. 2020.

[38] Kyle Kingsbury and Peter Alvaro. "Elle: Inferring Isolation Anomalies from Experimental Observations". In: *CoRR* abs/2003.10554 (2020). URL: https://arxiv.org/abs/2003.10554.

[39] Martin Kleppmann. *Hermitage: Testing transaction isolation levels*. https://github.com/ept/hermitage. 2020.

[40] LDBC. *Byelaws of the Linked Data Benchmark Council v1.1*. http://ldbcouncil.org/sites/default/files/ldbc-byelaws-1.1.pdf. 2017.

[41] Jure Leskovec et al. "Microscopic evolution of social networks". In: *KDD*. 2008, pp. 462–470. DOI: 10.1145/1401890.1401948.

[42] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. "Beyond Macrobenchmarks: Microbenchmark-based Graph Database Evaluation". In: *PVLDB* 12.4 (2018), pp. 390–403. URL: http://www.vldb.org/pvldb/vol12/p390-lissandrini.pdf.

[43] László Lőrincz et al. "Collapse of an online social network: Burning social capital to create it?" In: *Soc. Networks* 57 (2019), pp. 43–53. DOI: 10.1016/j.socnet.2018.11.004.

[44] Sara Magliacane, Alessandro Bozzon, and Emanuele Della Valle. "Efficient Execution of Top-K SPARQL Queries". In: *ISWC*. Springer, 2012, pp. 344–360. DOI: 10.1007/978-3-642-35176-1_22.

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

[46] Guido Moerkotte. "Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing". In: *PVLDB*. 1998, pp. 476–487. URL: http://www.vldb.org/conf/1998/p476.pdf.

[47] Mohamed Morsey et al. "DBpedia SPARQL Benchmark - Performance Assessment with Real Queries on Real Data". In: *ISWC*. 2011, pp. 454–469. DOI: 10.1007/978-3-642-25073-6_29.

[48] Seth A. Myers and Jure Leskovec. "The bursty dynamics of the Twitter information network". In: *WWW*. ACM, 2014, pp. 913–924. DOI: 10.1145/2566486.2568043.

[49] Lifeng Nai et al. "GraphBIG: Understanding graph computing in the context of industrial solutions". In: *SC*. 2015, 69:1–69:12. DOI: 10.1145/2807591.2807626.

[50] Thomas Neumann and Guido Moerkotte. "A Framework for Reasoning about Share Equivalence and Its Integration into a Plan Generator". In: *BTW*. 2009, pp. 7–26. URL: http://subs.emis.de/LNI/Proceedings/Proceedings144/article5220.html.

[51] Anil Pacaci et al. "Do We Need Specialized Graph Databases? Benchmarking Real-Time Social Networking Applications". In: *GRADES at SIGMOD*. 2017, 12:1–12:7. DOI: 10.1145/3078447.3078459.

[52] Andrew Pavlo and Matthew Aslett. "What's Really New with NewSQL?" In: *SIGMOD Rec.* (2016). DOI: 10.1145/3003665.3003674.

[53] Minh-Duc Pham, Peter A. Boncz, and Orri Erling. "S3G2: A Scalable Structure-Correlated Social Graph Generator". In: *TPCTC*. Vol. 7755. Springer, 2012, pp. 156–172. DOI: 10.1007/978-3-642-36727-4_11.

[54] Arnau Prat-Pérez. "LDBC SNB Datagen: Under the hood". In: *9th LDBC TUC Meeting*. 2017. URL: http://wiki.ldbcouncil.org/pages/viewpage.action?pageId=59277315&preview=/59277315/75431942/datagen_in_depth.pdf.

[55] Mark Raasveldt et al. "Fair Benchmarking Considered Difficult: Common Pitfalls In Database Performance Testing". In: *DBTest at SIGMOD*. ACM, 2018, 2:1–2:6. DOI: 10.1145/3209950.3209955.

[56] Liam Roditty. "Decremental maintenance of strongly connected components". In: *SODA*. SIAM, 2013, pp. 1143–1150. DOI: 10.1137/1.9781611973105.82.

[57] Liam Roditty and Uri Zwick. "On Dynamic Shortest Paths Problems". In: *ESA*. Vol. 3221. Lecture Notes in Computer Science. Springer, 2004, pp. 580–591. DOI: 10.1007/978-3-540-30140-0_52.

[58] Michael Schmidt et al. "SP^{2}Bench: A SPARQL Performance Benchmark". In: *Semantic Web Information Management - A Model-Based Perspective*. Springer, 2009, pp. 371–393. DOI: 10.1007/978-3-642-04329-1_16.

[59] Oszkár Semeráth et al. "Formal validation of domain-specific languages with derived features and well-formedness constraints". In: *Softw. Syst. Model.* 16.2 (2017), pp. 357–392. DOI: 10.1007/s10270-015-0485-x.

[60] Mirko Spasic, Milos Jovanovik, and Arnau Prat-Pérez. "An RDF Dataset Generator for the Social Network Benchmark with Real-World Coherence". In: *BLINK at ISWC*. 2016. URL: http://ceur-ws.org/Vol-1700/paper-02.pdf.

[61] Michael Stonebraker et al. “The End of an Architectural Era (It’s Time for a Complete Rewrite)”. In: *VLDB*. ACM, 2007, pp. 1150–1160. URL: http://www.vldb.org/conf/2007/papers/industrial/p1150-stonebraker.pdf.

[62] Gábor Szárnyas et al. “An early look at the LDBC Social Network Benchmark’s Business Intelligence workload”. In: *GRADES-NDA at SIGMOD/PODS*. ACM, 2018, 9:1–9:11. DOI: 10.1145/3210259.3210268.

[63] Gábor Szárnyas et al. “The Train Benchmark: Cross-technology performance evaluation of continuous model queries”. In: *Softw. Syst. Model.* 17.4 (2018), pp. 1365–1393. DOI: 10.1007/s10270-016-0571-8.

[64] TPC (Transaction Processing Performance Council). *TPC Benchmark C, revision 5.11*. 2010. URL: http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[65] TPC (Transaction Processing Performance Council). *TPC Pricing Specification, revision 2.7.0*. 2021. URL: http://tpc.org/TPC_Documents_Current_Versions/pdf/TPC-Pricing_v2.7.0.pdf.

[66] Johan Ugander et al. “The Anatomy of the Facebook Social Graph”. In: *CoRR* abs/1111.4503 (2011).

[67] Jack Waudby et al. “Preserving Reciprocal Consistency in Distributed Graph Databases”. In: *PaPoC at EuroSys*. ACM, 2020. DOI: 10.1145/3380787.3393675.

[68] Jack Waudby et al. “Supporting Dynamic Graphs and Temporal Entity Deletions in the LDBC Social Network Benchmark’s Data Generator”. In: *GRADES-NDA at SIGMOD*. ACM, 2020, 8:1–8:8. DOI: 10.1145/3398682.3399165.

[69] Jack Waudby et al. “Towards Testing ACID Compliance in the LDBC Social Network Benchmark”. In: *TPCTC*. Accepted. 2020.

A Choke Points

Introduction

Choke points are a superset of [9] with the exception of CP 7.1, which was removed and replaced with a new choke point. The correlations between choke points and queries are displayed in Table A.1.

	1.1	1.2	1.3	1.4	2.1	2.2	2.3	2.4	2.5	3.1	3.2	3.3	4.1	4.2	4.3	5.1	5.2	5.3	6.1	7.1	7.2	7.3	7.4	7.5	7.6	7.7	8.1	8.2	8.3	8.4	8.5	8.6
BI 1		⊗									⊗		⊗	⊗																	⊗	
BI 2								⊗		⊗	⊗		⊗	⊗	⊗			⊗	⊗										⊗			⊗	
BI 3	⊗	⊗	⊗		⊗	⊗		⊗				⊗																⊗				
BI 4		⊗	⊗		⊗	⊗	⊗	⊗				⊗						⊗	⊗									⊗		⊗		
BI 5		⊗					⊗																					⊗				
BI 6		⊗					⊗					⊗							⊗									⊗				
BI 7				⊗								⊗					⊗										⊗				
BI 8		⊗			⊗		⊗				⊗							⊗										⊗		⊗	⊗	
BI 9		⊗				⊗	⊗				⊗										⊗	⊗	⊗				⊗			⊗		
BI 10		⊗	⊗				⊗	⊗				⊗						⊗		⊗	⊗	⊗					⊗					⊗
BI 11	⊗						⊗		⊗																						
BI 12	⊗	⊗		⊗							⊗			⊗	⊗												⊗	⊗	⊗	⊗	⊗	
BI 13		⊗			⊗		⊗	⊗			⊗	⊗		⊗		⊗		⊗										⊗		⊗	⊗	
BI 14			⊗	⊗	⊗					⊗		⊗				⊗	⊗	⊗											⊗	⊗		
BI 15		⊗			⊗	⊗		⊗				⊗				⊗		⊗			⊗	⊗		⊗		⊗	⊗	⊗	⊗	⊗	⊗	⊗
BI 16																		⊗												⊗	⊗	
BI 17					⊗		⊗																				⊗					
BI 18									⊗																		⊗					
BI 19												⊗													⊗	⊗				⊗		⊗
BI 20												⊗													⊗	⊗				⊗		⊗
IC 1					⊗													⊗										⊗				
IC 2	⊗					⊗	⊗				⊗																				⊗	
IC 3					⊗					⊗						⊗												⊗			⊗	
IC 4							⊗																					⊗			⊗	
IC 5							⊗					⊗																⊗			⊗	
IC 6																⊗												⊗				
IC 7						⊗	⊗					⊗				⊗											⊗		⊗			
IC 8								⊗				⊗						⊗														
IC 9	⊗	⊗				⊗	⊗				⊗	⊗																			⊗	
IC 10							⊗					⊗	⊗	⊗		⊗	⊗		⊗	⊗											⊗
IC 11			⊗				⊗	⊗				⊗		⊗																		
IC 12												⊗									⊗	⊗						⊗				
IC 13												⊗									⊗	⊗		⊗			⊗				⊗
IC 14												⊗						⊗			⊗	⊗		⊗		⊗	⊗	⊗			⊗

Table A.1: Coverage of choke points by queries.

A.1 Aggregation Performance

CP-1.1: [QOPT] Interesting orders

TPC-H 1.2

This choke point tests the ability of the query optimizer to exploit the interesting orders induced by some operators. Apart from clustered indices providing key order, other operators also preserve or even induce tuple orderings. Sort-based operators create new orderings, typically the probe-side of a hash join conserves its order, etc.

Queries BI 3 BI 11 BI 12 IC 2 IC 9

CP-1.2: [QEXE] High cardinality group-by performance TPC-H 1.1

This choke point tests the ability of the execution engine to parallelize group-by's with a large number of groups. Some queries require performing large group-by's. In such a case, if an aggregation produces a significant number of groups, intra-query parallelization can be exploited as each thread may make its own partial aggregation. Then, to produce the result, these have to be re-aggregated. In order to avoid this, the tuples entering the aggregation operator may be partitioned by a hash of the grouping key and be sent to the appropriate partition. Each partition would have its own thread so that only that thread would write the aggregation, hence avoiding costly critical sections as well. A high cardinality distinct modifier in a query is a special case of this choke point. It is amenable to the same solution with intra-query parallelization and partitioning as the group-by. We further note that scale-out systems have an extra incentive for partitioning since this will distribute the CPU and memory pressure over multiple machines, yielding better platform utilization and scalability.

Queries BI 1 BI 3 BI 4 BI 5 BI 6 BI 8 BI 9 BI 10 BI 12 BI 13 BI 15 IC 9

CP-1.3: [QOPT] Top-k pushdown

This choke point tests the ability of the query optimizer to perform optimizations based on top-k selections. Many times queries demand for returning the top-k elements based on some property. Engines can exploit that once k results are obtained, extra restrictions in a selection can be added based on the properties of the kth element currently in the top-k, being more restrictive as the query advances, instead of sorting all elements and picking the highest k.

Queries BI 3 BI 4 BI 10 BI 14 IC 11

CP-1.4: [QEXE] Low cardinality group-by performance TPC-H 1.3

This choke point tests the ability to efficiently perform group-by evaluation when only a very limited set of groups is available. This can require special strategies for parallelization, e.g. pre-aggregation when possible. This case also allows using special strategies for grouping like using array lookup if the domain of keys is small.

Queries BI 7 BI 12 BI 14

A.2 Join Performance

CP-2.1: [QOPT] Rich join order optimization TPC-H 2.3

This choke point tests the ability of the query optimizer to find optimal join orders. A graph can be traversed in different ways. In the relational model, this is equivalent to different join orders. The execution time of these orders may differ by orders of magnitude. Therefore, finding an efficient join (traversal) order is important, which in general, requires enumeration of all the possibilities. The enumeration is complicated by operators that are not freely re-orderable like semi-, anti-, and outer-joins. Because of this difficulty most join enumeration algorithms do not enumerate all possible plans, and therefore can miss the optimal join order. Therefore, this choke point tests the ability of the query optimizer to find optimal join (traversal) orders.

Queries BI 3 BI 4 BI 8 BI 13 BI 14 BI 15 BI 17 IC 1 IC 3

CP-2.2: [QOPT] Late projection TPC-H 2.4

This choke point tests the ability of the query optimizer to delay the projection of unneeded attributes until late in the execution. Queries where certain columns are only needed late in the query. In such a situation, it is better to omit them from initial table scans, as fetching them later by row-id with a separate scan operator, which is

joined to the intermediate query result, can save temporal space, and therefore I/O. Late projection does have a trade-off involving locality, since late in the plan the tuples may be in a different order, and scattered I/O in terms of tuples/second is much more expensive than sequential I/O. Late projection specifically makes sense in queries where the late use of these columns happens at a moment where the amount of tuples involved has been considerably reduced; for example after an aggregation with only few unique group-by keys or a top-k operator.

Queries `BI 3` `BI 4` `BI 9` `BI 15` `IC 2` `IC 7` `IC 9`

CP-2.3: [QOPT] Join type selection

This choke point tests the ability of the query optimizer to select the proper join operator type, which implies accurate estimates of cardinalities. Depending on the cardinalities of both sides of a join, a hash or an index-based join operator is more appropriate. This is especially important with column stores, where one usually has an index on everything. Deciding to use a hash join requires a good estimation of cardinalities on both the probe and build sides. In TPC-H, the use of hash join is almost a foregone conclusion in many cases, since an implementation will usually not even define an index on foreign key columns. There is a break even point between index and hash based plans, depending on the cardinality on the probe and build sides.

Queries `BI 4` `BI 5` `BI 6` `BI 8` `BI 9` `BI 10` `BI 11` `BI 13` `BI 17` `IC 2` `IC 4` `IC 5` `IC 7` `IC 9` `IC 10` `IC 11`

CP-2.4: [QOPT] Sparse foreign key joins `TPC-H 2.2`

This choke point tests the performance of join operators when the join is sparse. Sometimes joins involve relations where only a small percentage of rows in one of the tables is required to satisfy a join. When tables are larger, typical join methods can be sub-optimal. Partitioning the sparse table, using Hash Clustered indices or implementing Bloom filter tests inside the join are techniques to improve the performance in such situations [27].

Queries `BI 2` `BI 3` `BI 4` `BI 10` `BI 13` `BI 15` `IC 8` `IC 11`

CP-2.5: [QEXE] Worst-case optimal joins

This choke point tests the query engine's ability to use multi-way joins to evaluate cyclic queries which are required to efficiently compute some dense subgraphs such as the triangle, the 4-cycle, and the diamond. The absence of multi-way joins (e.g. in traditional RDBMSs which only support binary joins), implies that join performance will be provably suboptimal.

Queries `BI 11` `BI 18`

A.3 Data Access Locality

CP-3.1: [QOPT] Detecting correlation `TPC-H 3.3`

This choke point tests the ability of the query optimizer to detect data correlations and exploiting them. If a schema rewards creating clustered indices, the question then is which of the date or data columns to use as key. In fact it should not matter which column is used, as range-propagation between correlated attributes of the same table is relatively easy. One way is through the creation of multi-attribute histograms after detection of attribute correlation. With MinMax indices, range-predicates on any column can be translated into qualifying tuple position ranges. If an attribute value is correlated with tuple position, this reduces the area to scan roughly equally to predicate selectivity.

Queries BI 2 BI 14 IC 3

CP-3.2: [STORAGE] Dimensional clustering

This choke point tests suitability of the identifiers assigned to entities by the storage system to better exploit data locality. A data model where each entity has a unique synthetic identifier, e.g. RDF or graph models, has some choice in assigning a value to this identifier. The properties of the entity being identified may affect this, e.g. type (label), other dependent properties, e.g. geographic location, date, position in a hierarchy, etc., depending on the application. Such identifier choice may create locality which in turn improves efficiency of compression or index access.

Queries BI 1 BI 2 BI 8 BI 9 BI 12 BI 13 IC 2 IC 9

CP-3.3: [QEXE] Scattered index access patterns

This choke point tests the performance of indices when scattered accesses are performed. The efficiency of index lookup is very different depending on the locality of keys coming to the indexed access. Techniques like vectoring non-local index accesses by simply missing the cache in parallel on multiple lookups vectored on the same thread may have high impact. Also detecting absence of locality should turn off any locality dependent optimizations if these are costly when there is no locality. A graph neighbourhood traversal is an example of an operation with random access without predictable locality.

Queries BI 3 BI 4 BI 6 BI 7 BI 10 BI 13 BI 14 BI 15 BI 19 BI 20 IC 5 IC 7 IC 8 IC 9 IC 10 IC 11 IC 12 IC 13 IC 14

A.4 Expression Calculation

CP-4.1: [QOPT] Common subexpression elimination TPC-H 4.2a

This choke point tests the ability of the query optimizer to detect common sub-expressions and reuse their results. A basic technique helpful in multiple queries is common subexpression elimination (CSE). CSE should recognize also that `avg` aggregates can be derived afterwards by dividing a `sum` by the `count` when those have been computed.

Queries BI 1 BI 2 IC 10

CP-4.2: [QOPT] Complex boolean expression joins and selections TPC-H 4.2d

This choke point tests the ability of the query optimizer to reorder the execution of boolean expressions to improve the performance. Some boolean expressions are complex, with possibilities for alternative optimal evaluation orders. For instance, the optimizer may reorder conjunctions to test first those conditions with larger selectivity [46].

Queries BI 1 BI 2 BI 12 BI 13 IC 10 IC 11

CP-4.3: [QEXE] Low overhead expressions interpretation

This choke point tests the ability of efficiently evaluating simple expressions on a large number of values. A typical example could be simple arithmetic expressions, mathematical functions like floor and absolute or date functions like extracting a year.

Queries BI 2 BI 12

A.5 Correlated Sub-queries

CP-5.1: [QOPT] Flattening sub-queries TPC-H 5.1

This choke point tests the ability of the query optimizer to flatten execution plans when there are correlated sub-queries. Many queries have correlated sub-queries and their query plans can be flattened, such that the correlated sub-query is handled using an equi-join, outer-join or anti-join. In TPC-H Q21, for instance, there is an `EXISTS` clause (for orders with more than one supplier) and a `NOT EXISTS` clauses (looking for an item that was received too late). To execute this query well, systems need to flatten both sub-queries, the first into an equi-join plan, the second into an anti-join plan. Therefore, the execution layer of the database system will benefit from implementing these extended join variants.

The ill effects of repetitive tuple-at-a-time sub-query execution can also be mitigated if execution systems by using vectorized, or blockwise query execution, allowing to run sub-queries with thousands of input parameters instead of one. The ability to look up many keys in an index in one API call creates the opportunity to benefit from physical locality, if lookup keys exhibit some clustering.

Queries BI 13 BI 14 BI 15 IC 3 IC 6 IC 7 IC 10

CP-5.2: [QEXE] Overlap between outer and sub-query TPC-H 5.3

This choke point tests the ability of the execution engine to reuse results when there is an overlap between the outer query and the sub-query. In some queries, the correlated sub-query and the outer query have the same joins and selections. In this case, a non-tree, rather DAG-shaped [50] query plan would allow to execute the common parts just once, providing the intermediate result stream to both the outer query and correlated sub-query, which higher up in the query plan are joined together (using normal query decorrelation rewrites). As such, the benchmark rewards systems where the optimizer can detect this and the execution engine supports an operator that can buffer intermediate results and provide them to multiple parent operators.

Queries BI 7 BI 14 IC 10

CP-5.3: [QEXE] Intra-query result reuse TPC-H 5.2

This choke point tests the ability of the execution engine to reuse sub-query results when two sub-queries are mostly identical. Some queries have almost identical sub-queries, where some of their internal results can be reused in both sides of the execution plan, thus avoiding to repeat computations.

Queries BI 2 BI 4 BI 8 BI 10 BI 13 BI 14 BI 15 BI 16 IC 1 IC 8 IC 14

A.6 Parallelism and Concurrency

CP-6.1: [QEXE] Inter-query result reuse TPC-H 6.3

This choke point tests the ability of the query execution engine to reuse results from different queries. Sometimes with a high number of streams a significant amount of identical queries emerge in the resulting workload. The reason is that certain parameters, as generated by the workload generator, have only a limited amount of parameters bindings. This weakness opens up the possibility of using a query result cache, to eliminate the repetitive part of the workload. A further opportunity that detects even more overlap is the work on recycling, which does not only cache final query results, but also intermediate query results of a "high worth". Here,

worth is a combination of partial-query result size, partial-query evaluation cost, and observed (or estimated) frequency of the partial-query in the workload.

Queries BI 2 BI 4 BI 6 IC 10

A.7 Graph Specifics

CP-7.1: [QEXE] Incremental path computation

This choke point tests the ability of the execution engine to reuse work across graph traversals. For example, when computing paths within a range of distances, it is often possible to incrementally compute longer paths by reusing paths of shorter distances that were already computed.

Queries BI 10 IC 10

CP-7.2: [QOPT] Cardinality estimation of transitive paths

This choke point tests the ability of the query optimizer to properly estimate the cardinality of intermediate results when executing transitive paths. A transitive path may occur in a "fact table" or a "dimension table" position. A transitive path may cover a tree or a graph, e.g. descendants in a geographical hierarchy vs. graph neighbourhood or transitive closure in a many-to-many connected social network. In order to decide proper join order and type, the cardinality of the expansion of the transitive path needs to be correctly estimated. This could for example take the form of executing on a sample of the data in the cost model or of gathering special statistics, e.g. the depth and fan-out of a tree. In the case of hierarchical dimensions, e.g. geographic locations or other hierarchical classifications, detecting the cardinality of the transitive path will allow one to go to a star schema plan with scan of a fact table with a selective hash join. Such a plan will be on the other hand very bad for example if the hash table is much larger than the "fact table" being scanned.

Queries BI 9 BI 10 BI 15 IC 12 IC 13 IC 14

CP-7.3: [QEXE] Execution of a transitive step

This choke point tests the ability of the query execution engine to efficiently execute transitive steps. Graph workloads may have transitive operations, for example finding a shortest path between nodes. This involves repeated execution of a short lookup, often on many values at the same time, while usually having an end condition, e.g. the target node being reached or having reached the border of a search going in the opposite direction. For the best efficiency, these operations can be merged or tightly coupled to the index operations themselves. Also parallelization may be possible but may need to deal with a global state, e.g. set of visited nodes. There are many possible tradeoffs between generality and performance.

Queries BI 9 BI 10 BI 15 IC 12 IC 13 IC 14

CP-7.4: [QEXE] Efficient evaluation of termination criteria for transitive queries

This tests the ability of a system to express termination criteria for transitive queries so that not the whole transitive relation has to be evaluated as well as efficient testing for termination.

Queries BI 9

CP-7.5: [QEXE] Unweighted shortest paths

A common problem in graph queries is determining the distance between a node and a set of nodes. To compute the distance values, systems may employ BFS or a single-source shortest path algorithm with uniform weights. To compute the distance between two given node, systems can use bidirectional search algorithms.

Queries BI 15 IC 13 IC 14

CP-7.6: [QEXE] Weighted shortest paths

Computing single-source shortest path is a fundamental problem in graph queries. While there are well-known algorithms to compute it, e.g. Dijkstra's algorithm or the Bellman-Ford algorithm, system often use naïve approaches such as enumerating all paths which makes these queries intractable.

Queries BI 19 BI 20

CP-7.7: [QEXE] Composition of graph queries

In many cases, it is desirable to specify multiple graph queries, where the first one defines an induced subgraph or an overlay graph on the original graph, which is then passed two the next query, and so on. Expressing such computations as a sequence of composable graph queries would be desirable from both usability, optimization, and execution aspects. However, currently many graph dabases lack support for composable graph queries.

The G-CORE [5] design language tackled problem this by introducing the *path property graph* data model (consisting of nodes, edges, and paths) and defining queries such that they return a graph (while also providing means to return a tabular output).

Queries BI 15 BI 19 BI 20 IC 14

A.8 Language Features

CP-8.1: [LANG] Complex patterns

Description A natural requirement for graph query systems is to be able to express complex graph patterns.

Transitive edges. Transitive closure-style computations are common in graph query systems, both with fixed bounds (e.g. get nodes that can be reached through at least 3 and at most 5 knows edges), and without fixed bounds (e.g. get all Messages that a Comment replies to).

Negative edge conditions. Some queries define *negative pattern conditions*. For example, the condition that a certain Message does not have a certain Tag is represented in the graph as the absence of a hasTag edge between the two nodes. Thus, queries looking for cases where this condition is satisfied check for negative patterns, also known as negative application conditions (NACs) in graph transformation literature [31].

Language-specific notes Negative edge conditions are often difficult to express in early stage graph query languages. Notably, this is showcased by the fact that early versions of both the SPARQL and Cypher languages used cumbersome syntax to express such conditions.

Cypher. Prior to Neo4j version 2.0, Cypher queries used a syntax such as the following:

```
MATCH (source)-[r?:someType]->(target)
WHERE r IS NULL
RETURN source
```

In Neo4j 2.0, the `OPTIONAL MATCH` clause was introduced:[1]

```
MATCH (source)
OPTIONAL MATCH (source)-[r:someType]->(target)
WHERE r IS NULL
RETURN source
```

However, the preferred method is to use a pattern condition in the `WHERE` clause:

```
MATCH (source)
WHERE NOT (source)-[:someType]->(target)
RETURN source
```

SPARQL. Prior to SPARQL 1.1, queries with negative edge conditions could be expressed with the following approach:

```
OPTIONAL {
  ?xRoute ?routeDefinition ?xSensor .
  FILTER (sameTerm(base:routeDefinition, routeDefinition))
} .
FILTER (!bound(?routeDefinition))
```

Since SPARQL 1.1, the preferred method is using the `NOT EXISTS` construct.

```
?xSensor rdf:type base:Sensor .
?xRoute base:switchPosition ?xSwitchPosition .
FILTER NOT EXISTS { xRoute ?routeDefinition ?xSensor } .
```

The `MINUS` construct can also be used for defining a negative condition for a single edge.[2]

```
?xSensor rdf:type base:Sensor .
?xRoute base:switchPosition ?xSwitchPosition .
MINUS { xRoute ?routeDefinition ?xSensor } .
```

Queries BI 7 BI 9 BI 10 BI 12 BI 15 BI 17 BI 18 IC 7 IC 13 IC 14

CP-8.2: [LANG] Complex aggregations

Description BI workloads are heavy on aggregation, including queries with *subsequent aggregations*, where the results of an aggregation serves as the input of another aggregation. Expressing such operations requires some sort of query composition or chaining (see also CP-8.4). It is also common to *filter on aggregation results* (similarly to the `HAVING` keyword of SQL).

Language-specific notes

Cypher. Cypher does not allow aggregations on aggregations, e.g. `avg(count(x))`, as the semantics of such expressions is not meaningful. However, it allows multiple aggregations is subsequent `WITH` and `RETURN` clauses. There were multiple discussion rounds on the aggregation semantics of the openCypher language.[3]

SPARQL. SPARQL requires users to explicitly enumerate variables in the `GROUP BY` clause (similarly to SQL).

Queries BI 2 BI 3 BI 4 BI 5 BI 6 BI 8 BI 12 BI 13 BI 15 IC 1 IC 3 IC 4 IC 5 IC 6 IC 12 IC 14

[1] https://dzone.com/articles/new-neo4j-optional

[2] For details, see the "Relationship and differences between `NOT EXISTS` and `MINUS`" section in the SPARQL 1.1 specification at https://www.w3.org/TR/sparql11-query/#neg-notexists-minus

[3] http://www.opencypher.org/blog/2017/07/27/ocig1-aggregations-blog/

CP-8.3: [LANG] Ranking-style queries

Description Additionally to aggregations, BI workloads often use *window functions*, which perform aggregations without grouping input tuples to a single output tuple. A common use case for windowing is *ranking*, i.e. selecting the top element with additional values in the tuple (nodes, edges or attributes).[4]

Language-specific notes

Cypher. Ranking can be expressed in Cypher as a sequence of ordering, collecting results and taking the top-k values of the result list.

```
...
WITH x, ...
ORDER BY x
WITH collect(x) AS xs
WITH xs[0] AS top, xs[0..5] AS top5
...
```

SPARQL. SPARQL 1.0 allows simple top-k queries. SPARQL 1.1 introduced scoring functions that can define a variable to be used for ordering [44]. *Window functions* are not yet supported but are under consideration for SPARQL 1.2[5].

Queries BI 12 BI 14 BI 15 IC 7 IC 14

CP-8.4: [LANG] Query composition

Description Numerous use cases require *composition* of queries, including the reuse of query results (e.g. nodes, edges) or using scalar subqueries (e.g. selecting a threshold value with a subquery and using it for subsequent filtering operations).

Language-specific notes

Cypher. Nested subqueries were accepted in the openCypher language.[6]. Neo4j 4.1 introduced support for correlated subqueries.

SPARQL. SPARQL fully supports query composition.[7]

Queries BI 4 BI 8 BI 12 BI 13 BI 14 BI 15 BI 16 BI 19 BI 20

CP-8.5: [LANG] Dates and times

Description Handling dates and times is a fundamental requirement for production-ready database systems. It is particularly important in the context of BI queries as these often calculate aggregations on certain periods of time (e.g. on entities created during the course of a month).

Language-specific notes

[4] PostgreSQL defines the `OVER` keyword to use aggregation functions as window functions, and the `rank()` function to produce numerical ranks, see `https://www.postgresql.org/docs/9.1/static/tutorial-window.html` for details.

[5] `https://github.com/w3c/sparql-12/issues/47`

[6] `https://github.com/petraselmer/openCypher/blob/1ca70bf8e3cea65ee47ce49eeabea83530eb529b/cip/1.accepted/CIP2016-06-22-nested-updating-and-chained-subqueries.adoc`

[7] `https://www.w3.org/TR/sparql11-query/#subqueries`

Cypher. The openCypher project has accepted a proposal to support dates and times.[8] Neo4j has datetime support since version 3.4.

SPARQL. SPARQL supports dates and times extensively with timezones and functions for extracting a part of a given date.[9]

Queries BI 1 BI 2 BI 8 BI 9 BI 12 BI 13 BI 15 BI 16 IC 2 IC 3 IC 4 IC 5 IC 9

CP-8.6: [LANG] Handling paths

Description

Note on terminology. The *Glossary of graph theory terms* page of Wikipedia[10] defines *paths* as follows: "A path may either be a walk (a sequence of nodes and edges, with both endpoints of an edge appearing adjacent to it in the sequence) or a simple path (a walk with no repetitions of nodes or edges), depending on the source." In this work, we use the first definition, which is more common in modern graph database systems and is also followed in a recent survey on graph query languages [4].

Handling paths as first-class citizens is one of the key distinguishing features of graph database systems [5]. Hence, additionally to reachability-style checks, a language should be able to express queries that operate on elements of a path, e.g. calculate a score on each edge of the path. Also, some use cases specify uniqueness constraints on paths [4]: *arbitrary path*, *shortest path*, *no-repeated-node semantics* (also known as *simple paths*), and *no-repeated-edge semantics* (also known as *trails*). Other variants are also used in rare cases, such as *maximal* (non-expandable) or *minimal* (non-contractable) paths.

Language-specific notes

Cypher. Cypher uses *no-repeated-edge matching semantics* (in return, this semantics is sometimes dubbed as *cyphermorphism*). Configurable matching semantics (e.g. `MATCH ALL WALKS`) were proposed in the openCypher language.[11] Regular path queries (RPQs) are also proposed in the openCypher language as *path patterns*.[12]

SPARQL. SPARQL uses *homomorphism-based matching semantics* and supports RPQs as *property paths*. Isomorphism-based matching semantics can be expressed by introducing custom filtering condition, e.g. `FILTER ( ?e1 != ?e2 ).`

G-CORE. The G-CORE language [5] treats paths as *first-order citizens*: its *path property graph data model* can store paths in the graph model with labels and properties. However, it only supports shortest path semantics (for tractability reasons) and does not allow enumeration of all paths. G-CORE uses *homomorphism-based matching semantics*.

Queries BI 10 BI 15 BI 19 BI 20 IC 10 IC 13 IC 14

A.9 Refresh operations

CP-9.1: [REF] Insert node

This choke point tests the ability of the database to insert a node.

[8] https://github.com/thobe/openCypher/blob/06accdb3e69820cdfac3dbd50a4f8eee73ba179a/cip/1.accepted/CIP2015-08-06-date-time.adoc

[9] https://www.w3.org/TR/sparql11-query/#func-date-time

[10] https://en.wikipedia.org/wiki/Glossary_of_graph_theory_terms

[11] https://github.com/boggle/openCypher/blob/c139130b49aebe6a85fc395e2cf03cfeec8484c6/cip/1.accepted/CIP2017-01-18-configurable-pattern-matching-semantics.adoc

[12] https://github.com/thobe/openCypher/blob/b95eec108ce4ec07eedfe13b3e5fff0e94f789a4/cip/1.accepted/CIP2017-02-06-Path-Patterns.adoc

Queries `INS 1` `INS 4` `INS 5` `INS 6` `INS 7`

CP-9.2: [REF] Insert edge

This choke point tests the ability of the database to insert an edge.

Queries `INS 1` `INS 2` `INS 3` `INS 4` `INS 5` `INS 6` `INS 7` `INS 8`

CP-9.3: [REF] Delete node

This choke point tests the ability of the database to delete a node.

Queries `DEL 1` `DEL 4` `DEL 6` `DEL 7`

CP-9.4: [REF] Delete edge

This choke point tests the ability of the database to delete an edge.

Queries `DEL 1` `DEL 2` `DEL 3` `DEL 4` `DEL 5` `DEL 6` `DEL 7` `DEL 8`

CP-9.5: [REF] Delete recursively

This choke point tests the ability of the database to recursively perform a delete operation, e.g. delete an entire message thread.

Queries `DEL 1` `DEL 4` `DEL 6` `DEL 7`

B Scale Factor Statistics

B.1 Number of Entities

C	File	SF0.1	SF0.3	SF1	SF3	SF10	SF30	SF100	SF300	SF1 000
N	organisation	7 955	7 955	7 996	7 996	7 996	7 996	7 996	7 996	7 996
E	organisation_isLocatedIn_place	7 955	7 955	7 996	7 996	7 996	7 996	7 996	7 996	7 996
N	place	1 460	1 460	1 466	1 466	1 466	1 466	1 466	1 466	1 466
E	place_isPartOf_place	1 454	1 454	1 460	1 460	1 460	1 460	1 460	1 460	1 460
N	tag	16 080	16 080	16 080	16 080	16 080	16 080	16 080	16 080	16 080
E	tag_hasType_tagclass	16 080	16 080	16 080	16 080	16 080	16 080	16 080	16 080	16 080
N	tagclass	71	71	71	71	71	71	71	71	71
E	tagclass_isSubclassOf_tagclass	70	70	70	70	70	70	70	70	70
N	comment	203 354	682 061	2 343 952	7 135 636	24 271 888	73 590 941	243 266 898	710 752 235	2 335 637 135
E	comment_hasCreator_person	203 354	682 061	2 343 952	7 135 636	24 271 888	73 590 941	243 266 898	710 752 235	2 335 637 135
E	comment_hasTag_tag	232 524	807 266	3 069 162	17 465 623	605414570	96 053 813	317 369 562	926 124 724	3 042 978 961
E	comment_isLocatedIn_place	203 354	682 061	2 343 952	7 135 636	24 271 888	73 590 941	243 266 898	710 752 235	2 335 637 135
E	comment_replyOf_comment	103 552	346 553	1 187 815	3 619 711	12 306 670	37 324 357	123 386 519	360 517 003	1 184 778 982
E	comment_replyOf_post	99 802	335 508	1 156 137	3 515 925	11 965 218	36 266 584	119 880 379	350 235 232	1 150 858 153
N	forum	16 818	38 050	110 202	272 268	729 153	1 842 141	5 002 291	12 561 079	36 098 481
E	forum_containerOf_post	168 873	404 531	1 214 766	3 140 119	8 915 649	23 765 756	68 871 360	182 980 982	555 306 166
E	forum_hasMember_person	266 965	861 079	3 260 578	9 939 453	33 883 607	103901443	341 232 279	995 330 706	3 277 239 057
E	forum_hasModerator_person	16 818	38 050	110 202	272 268	729 153	1 842 141	5 002 291	12 561 079	36 098 481
E	forum_hasTag_tag	54 288	124 186	355 354	16 205 018	2 369 727	5 976 729	16 195 463	40 653 342	116 727 525
N	person	1 700	3 900	11 000	27 000	73 000	184 000	499 000	1 254 000	3 600 000
A	person_email_emailaddress	3 690	8 393	18 602	45 573	124 555	312 925	850 804	2 140 338	6 141 306
E	person_hasInterest_tag	39 170	90 036	256 152	628 563	1 713 574	4 318 588	11 692 172	29 346 263	84 229 044
E	person_isLocatedIn_place	1 700	3 900	11 000	27 000	73 000	184 000	499 000	1 254 000	3 600 000
E	person_knows_person	18 074	57 179	452 622	1 370 174	4 654 416	14 212 356	46 598 276	136 219 368	447 163 916
E	person_likes_comment	96 865	412 010	1 649 394	5 555 074	21 418 614	71 641 419	260 701 994	820 056 009	2 858 070 323
E	person_likes_post	97 638	328 473	1 170 372	3 629 288	12 661 782	39 694 513	135 205 141	404 808 353	1 361 722 197
A	person_speaks_language	3 771	8 595	24 204	59 467	160 779	405 403	1 099 440	2 763 075	7 932 926
E	person_studyAt_organisation	1 337	3 089	8 820	21 574	58 429	147 005	398 560	1 002 380	2 878 718
E	person_workAt_organisation	3 732	8 561	23 969	58 843	158 961	401 356	1 086 037	2 728 559	7 829 672
N	post	168 873	404 531	1 214 766	3 140 119	8 915 649	23 765 756	68 871 360	182 980 982	555 306 166
E	post_hasCreator_person	168 873	404 531	1 214 766	3 140 119	8 915 649	23 765 756	68 871 360	182 980 982	555 306 166
E	post_hasTag_tag	59 862	207 814	789 735	2 384 629	8 216 364	24 931 521	82 466 083	241 151 541	793 254 841
E	post_isLocatedIn_place	168 873	404 531	1 214 766	3 140 119	8 915 649	23 765 756	68 871 360	182 980 982	555 306 166

Table B.1: The number of entries per SF and per file in the Interactive workload (produced by the Hadoop-based generator and measured based on the output of the CsvBasic serializer). To derive these numbers, 100% of the network was generated as an initial bulk data set with no update streams. Notation – C: entity category, N: node, E: edge, A: attribute.

C	File	SF0.1	SF0.3	SF1	SF3	SF10	SF30	SF100	SF300	SF1 000

Table B.2: The number of entries per SF and per file in the BI workload (produced by the Spark-based generator. TODO). Notation – C: entity category, N: node, E: edge, A: attribute.

C Benchmark Checklist

We expect LDBC benchmarks to be used in many scenarios. For most research papers, fully audited results are unrealistic and even unaudited results can provide insight into the performance of the systems under test (SUT). However, we ask authors to include the following information in their papers:

- Were the results cross-validated for at least one scale factor?
- Were the results cross-validated for all scale factors used in the benchmark?
- Does the SUT have a persistent storage?
- Does the SUT provide ACID transactions?
- Does the SUT provide any level of fault-tolerance?
- How many warm-up rounds were performed?
- How many execution rounds were performed?
- How were the execution times summarized?[1]
- Is the loading phase included in the query execution times?[2]
- If the SUT is not your own system, did you contact its developers or experts to help optimizing the queries?[3]

These results will help the reader to put the results in context. For example, a non-ACID compliant, non-fault-tolerant system working on read-only graphs and offering no persistent storage is expected to have significantly better results than a fully-fledged disk-based DBMS.

We also suggest the reader to take a look at the checklist presented in [55].

[1] Paper [32] provides an excellent overview on how to summarize benchmark results.

[2] This might be relevant for systems without persistent storage, or systems providing lazy/incremental computation.

[3] For a research prototype tool, the tuning knobs are usually not well documented. Hence, it is worth contacting the tool's authors, who are generally keen to help. For more mature systems (e.g. most established RDBMSs), there is a large body of knowledge available, in the form of books and online forums, which should help your optimization efforts. It is also possible to contact experienced DBAs who can assist with fine tuning the system.

D Legacy Data Sets for the Interactive workload

The Interactive workload uses the legacy version of the data sets. These can be generated using the Hadoop-based Datagen hosted on the **stable** branch of the repositories. This chapter documents these data sets.

The data sets will be hosted publicly soon.

The **key differences from the latest (BI) data sets** are the following:

- DateTime values follow the format `yyyy-mm-ddTHH:MM:ss.sss+0000`, i.e. their offset string is `0000` instead of `00:00`. This implies that they are not compatible with the recommendations of RFC-3339[1].
- The Forum-hasModerator-Person edge type has an *exactly one* cardinality on the Person's end.

D.1 Output Data

For each scale factor, Datagen produces three different artefacts:

- **Dataset:** The dataset to be bulk loaded by the SUT. In the Interactive workload, it corresponds to roughly the 90% of the total generated network.
- **Update Streams:** A set of update streams containing update queries, which are used by the driver to generate the update queries of the workloads. This update streams correspond to the remaining 10% of the generated dataset.
- **Substitution Parameters:** A set of files containing the different parameter bindings that will be used by the driver to generate the read queries of the workloads.

D.1.1 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 Gibibytes of the generated output files using the CsvMergeForeign serializer (see Section D.1.2) and default settings, i.e. both the 90% initial data and the 10% update streams count towards the total size. For example, SF1 takes roughly 1 GiB in CSV format, SF3 weighs roughly 3 GiB and so on and so forth. It is important to note that for a given scale factor, data sets generated using different serializers contain exaclty the same data, the only difference is in how they are represented.[2] The provided SFs are the following: 1, 3, 10, 30, 100, 300, 1000. Additionally, two small data sets, 0.1, and 0.3 are provided to help initial validation efforts.

The Test Sponsor may select the SF that better fits their needs, by properly configuring the Datagen, as described in Section 3.3. The size of the resulting dataset is mainly affected by the following configuration parameters: the number of persons and the number of years simulated. By default, all SFs are defined over a period of three years, starting from 2010, and SFs are computed by scaling the number of Persons in the network. Table D.1 shows some metrics of SFs 0.1, ..., 1000 data sets.

Scale Factor	**0.1**	**0.3**	**1**	**3**	**10**	**30**	**100**	**300**	**1 000**
# Persons	1.5K	3.5K	11K	27K	73K	182K	499K	1.25M	3.6M
# of nodes	327.6K	908K	3.2M	9.3M	30M	88.8M	282.6M	817.3M	2.7B
# of edges	1.5M	4.6M	17.3M	52.7M	176.6M	540.9M	1.8B	5.3B	17B

Table D.1: Properties of data sets for each scale factor in the Interactive workload produced by the Hadoop-based generator. For detailed statistics, see Table B.1

[1] `https://tools.ietf.org/html/rfc3339`

[2] Naturally, there are slight differences in the disk usage of the data sets created with different serializers. For example, for a given scale factor, the disk usage of the data set serialized with the CsvBasic serializer is expected to be higher, while with the CsvMergeForeignComposite, it is expected to be lower.

Serializer	Nodes	Attributes		Edges	
		single-valued	multi-valued	one-to-many	many-to-many
CsvBasic	⊗	○	⊗	⊗	⊗
CsvComposite	⊗	○	○	⊗	⊗
CsvMergeForeign	⊗	○	⊗	○	⊗
CsvCompositeMergeForeign	⊗	○	○	○	⊗

Table D.2: Attributes and edges serialized to separate files the different CSV serializers.

Table D.2 show how each CSV serializer handles attributes/edges of different cardinalities. The data shows why CsvBasic has the most files and CsvCompositeMergeForeign has the least number of files.

C	File	Content
N	organisation_*.csv	id | type | name | url
E	organisation_isLocatedIn_place_*.csv	Organisation.id | Place.id
N	place_*.csv	id | name | url | type
E	place_isPartOf_place_*.csv	Place.id | Place.id
N	tag_*.csv	id | name | url
E	tag_hasType_tagclass_*.csv	Tag.id | TagClass.id
N	tagclass_*.csv	id | name | url
E	tagclass_isSubclassOf_tagclass_*.csv	TagClass.id | TagClass.id
N	comment_*.csv	creationDate | id | locationIP | browserUsed | content | length
E	comment_hasCreator_person_*.csv	creationDate | Comment.id | Person.id
E	comment_hasTag_tag_*.csv	creationDate | Comment.id | Tag.id
E	comment_isLocatedIn_place_*.csv	creationDate | Comment.id | Place.id
E	comment_replyOf_comment_*.csv	creationDate | Comment.id | ParentComment.id
E	comment_replyOf_post_*.csv	creationDate | Comment.id | ParentPost.id
N	forum_*.csv	creationDate | id | title | type
E	forum_containerOf_post_*.csv	creationDate | Forum.id | Post.id
E	forum_hasMember_person_*.csv	creationDate | Forum.id | Person.id | type
E	forum_hasModerator_person_*.csv	creationDate | Forum.id | Person.id
E	forum_hasTag_tag_*.csv	creationDate | Forum.id | Tag.id
N	person_*.csv	creationDate | id | firstName | lastName | gender | birthday | locationIP | browserUsed
A	person_email_emailaddress_*.csv	creationDate | Person.id | email
E	person_hasInterest_tag_*.csv	creationDate | Person.id | Tag.id
E	person_isLocatedIn_place_*.csv	creationDate | Person.id | Place.id
E	person_knows_person_*.csv	creationDate | Person1.id | Person2.id
E	person_likes_comment_*.csv	creationDate | Person.id | Comment.id
E	person_likes_post_*.csv	creationDate | Person.id | Post.id
A	person_speaks_language_*.csv	creationDate | Person.id | language
E	person_studyAt_organisation_*.csv	creationDate | Person.id | Organisation.id | classYear
E	person_workAt_organisation_*.csv	creationDate | Person.id | Organisation.id | workFrom
N	post_*.csv	creationDate | id | imageFile | locationIP | browserUsed | language | content | length | Forum.id
E	post_hasCreator_person_*.csv	creationDate | Post.id | Person.id
E	post_hasTag_tag_*.csv	creationDate | Post.id | Tag.id
E	post_isLocatedIn_place_*.csv	creationDate | Post.id | Place.id

Table D.3: Files output by the CsvBasic serializer (33 in total). The first part of the table contains the static entites, the second part contains the dynamic ones. Notation – C: entity category, N: node, E: edge, A: attribute.

D.1.2 Serializers

The datasets are generated in the `social_network/` directory, split into static and dynamic parts (Figure 3.1). The filenames (without the extension) end in `_i_j` where `i` is the block id and `j` is the partition id (set by `numThreads`). The SUT has to take care only of the generated Dataset to be bulk loaded. Using `NULL` values for storing optional values is allowed.

C	File	Content
N	organisation_*.csv	id | type | name | url | place
N	place_*.csv	id | name | url | type | isPartOf
N	tag_*.csv	id | name | url | hasType
N	tagclass_*.csv	id | name | url | isSubclassOf
N	comment_*.csv	id | creationDate | locationIP | browserUsed | content | length | creator | place | replyOfPost | replyOfComment
E	comment_hasTag_tag_*.csv	Comment.id | Tag.id
N	forum_*.csv	id | title | creationDate | moderator
E	forum_hasMember_person_*.csv	Forum.id | Person.id | joinDate/creationDate
E	forum_hasTag_tag_*.csv	Forum.id | Tag.id
N	person_*.csv	id | firstName | lastName | gender | birthday | creationDate | locationIP | browserUsed | place
A	person_email_emailaddress_*.csv	Person.id | email
E	person_hasInterest_tag_*.csv	Person.id | Tag.id
E	person_knows_person_*.csv	Person.id | Person.id | creationDate
E	person_likes_comment_*.csv	Person.id | Post.id | creationDate
E	person_likes_post_*.csv	Person.id | Post.id | creationDate
A	person_speaks_language_*.csv	Person.id | language
E	person_studyAt_organisation_*.csv	Person.id | Organisation.id | classYear
E	person_workAt_organisation_*.csv	Person.id | Organisation.id | workFrom
N	post_*.csv	id | imageFile | creationDate | locationIP | browserUsed | language | content | length | creator | Forum.id | place
E	post_hasTag_tag_*.csv	Post.id | Tag.id

Table D.4: Files output by the CsvMergeForeign serializer (20 in total). The first part of the table contains the static entites, the second part contains the dynamic ones. Notation – C: entity category, T: entity type, N: node, E: edge, A: attribute.

Datagen currently only supports CSV-based serializers. These produce CSV-like text files which use the pipe character "|" as the primary field separator and the semicolon character ";" as a separator for multi-valued attributes (for the composite serializers). The CSV files are stored in two subdirectories: `static/` and `dynamic/`. Depending on the number of threads used for generating the dataset, the number of files varies, since there is a file generated per thread. We indicate with "$*$" in the specification.

The following CSV variants are supported:

- **CsvBasic:** Each entity, relation and attribute with a cardinality larger than one (including attributes `Person.email` and `Person.speaks`), are output in a separate file. Generated files are summarized in Table D.3.
- **CsvMergeForeign:** This serializer is similar to CsvBasic, but relations that have a cardinality of 1-to-N are merged in the entity files as a foreign keys. There are 13 such relations in total:
  - comment_hasCreator_person, comment_isLocatedIn_place, comment_replyOf_comment, comment_replyOf_post (merged to Comment);
  - forum_hasModerator_person (merged to Forum);
  - organisation_isLocatedIn_place (merged to Organisation);
  - person_isLocatedIn_place (merged to Person);
  - place_isPartOf_place (merged to Place);
  - post_hasCreator_person, post_isLocatedIn_place, forum_containerOf_post (merged to Post);
  - tag_hasType_tagclass (merged to Tag);
  - tagclass_isSubclassOf_tagclass (merged to TagClass)

  Generated files are summarized in Table D.4.
- **CsvComposite:** Similar to the CsvBasic format but each entity, and relations with a cardinality larger than one, are output in a separate file. Multi-valued attributes (`Person.email` and `Person.speaks`) are stored as composite values. Generated files are summarized in Table D.5.
- **CsvCompositeMergeForeign:** Has the traits of both the CsvComposite and the CsvMergeForeign formats. Multi-valued attributes (`Person.email` and `Person.speaks`) are stored as composite values. Generated files are summarized in Table D.6.
- **CsvBasic using "raw data" mode:** The file names are the same as in CsvBasic but there are two important differences: (1) additional attributes, e.g. `deletionDate`, `explicitlyDeleted`, and `weight` (used for weighted

C	File	Content
N	organisation_*.csv	id | type | name | url
E	organisation_isLocatedIn_place_*.csv	Organisation.id | Place.id
N	place_*.csv	id | name | url | type
E	place_isPartOf_place_*.csv	Place.id | Place.id
N	tag_*.csv	id | name | url
E	tag_hasType_tagclass_*.csv	Tag.id | TagClass.id
N	tagclass_*.csv	id | name | url
E	tagclass_isSubclassOf_tagclass_*.csv	TagClass.id | TagClass.id
N	comment_*.csv	id | creationDate | locationIP | browserUsed | content | length
E	comment_hasCreator_person_*.csv	Comment.id | Person.id
E	comment_hasTag_tag_*.csv	Comment.id | Tag.id
E	comment_isLocatedIn_place_*.csv	Comment.id | Place.id
E	comment_replyOf_comment_*.csv	Comment.id | Comment.id
E	comment_replyOf_post_*.csv	Comment.id | Post.id
N	forum_*.csv	id | title | creationDate
E	forum_containerOf_post_*.csv	Forum.id | Post.id
E	forum_hasMember_person_*.csv	Forum.id | Person.id | joinDate/creationDate
E	forum_hasModerator_person_*.csv	Forum.id | Person.id
E	forum_hasTag_tag_*.csv	Forum.id | Tag.id
N	person_*.csv	id | firstName | lastName | gender | birthday | creationDate | locationIP | browserUsed | language | email
E	person_hasInterest_tag_*.csv	Person.id | Tag.id
E	person_isLocatedIn_place_*.csv	Person.id | Place.id
E	person_knows_person_*.csv	Person.id | Person.id | creationDate
E	person_likes_comment_*.csv	Person.id | Post.id | creationDate
E	person_likes_post_*.csv	Person.id | Post.id | creationDate
E	person_studyAt_organisation_*.csv	Person.id | Organisation.id | classYear
E	person_workAt_organisation_*.csv	Person.id | Organisation.id | workFrom
N	post_*.csv	id | imageFile | creationDate | locationIP | browserUsed | language | content | length
E	post_hasCreator_person_*.csv	Post.id | Person.id
E	post_hasTag_tag_*.csv	Post.id | Tag.id
E	post_isLocatedIn_place.csv	Post.id | Place.id

Table D.5: Files output by the CsvComposite serializer (31 in total). The first part of the table contains the static entites, the second part contains the dynamic ones. Notation – C: entity category, N: node, E: edge.

graphs in Graphalytics [34]), are included mainly for debugging, (2) all data is included, i.e. if a Forum is created and deleted before sampling the initial data set, it is included in this data set. Generated files are summarized in Table D.7.

The inheritance hierarchies in the schema have two important characteristics (1) all subclasses use the same id space, e.g. there cannot be a Comment and a Post with id 1 at the same time, (2) they are serialized to CSVs using either the *map hierarchy to single table* or *map each concrete class to its own table* strategies[3]:

Message = Comment | Post *Map each concrete class to its own table* is used i.e. separate CSV files are used for the Comment and the Post classes.

Place = City | Country | Continent *Map hierarchy to single table* is used, i.e. all Place node are serialized in a single file. A discriminator attribute "type" is used with the value denoting the concrete class, e.g. "Country".

Organisation = Company | University *Map hierarchy to single table* is used, i.e. all Organisation nodes are serialized in a single fiel. A discriminator attribute "type" is used with the value denoting the concrete class, e.g. "Company".

D.1.3 Update Streams

The generic schema is given in Table D.8, while the concrete schema of each insert operation is given in Table D.9. The update stream files are generated in the `social_network/` directory and are grouped as follows:

[3] `http://www.agiledata.org/essays/mappingObjects.html`

C	File	Content
N	organisation_*.csv	id | type | name | url | place
N	place_*.csv	id | name | url | type | isPartOf
N	tag_*.csv	id | name | url | hasType
N	tagclass_*.csv	id | name | url | isSubclassOf
N	comment_*.csv	id | creationDate | locationIP | browserUsed | content | length | creator | place | replyOfPost | replyOfComment
E	comment_hasTag_tag_*.csv	Comment.id | Tag.id
N	forum_*.csv	id | title | creationDate | moderator
E	forum_hasMember_person_*.csv	Forum.id | Person.id | joinDate/creationDate
E	forum_hasTag_tag_*.csv	Forum.id | Tag.id
N	person_*.csv	id | firstName | lastName | gender | birthday | creationDate | locationIP | browserUsed | place | language | email
E	person_hasInterest_tag_*.csv	Person.id | Tag.id
E	person_knows_person_*.csv	Person.id | Person.id | creationDate
E	person_likes_comment_*.csv	Person.id | Post.id | creationDate
E	person_likes_post_*.csv	Person.id | Post.id | creationDate
E	person_studyAt_organisation_*.csv	Person.id | Organisation.id | classYear
E	person_workAt_organisation_*.csv	Person.id | Organisation.id | workFrom
N	post_*.csv	id | imageFile | creationDate | locationIP | browserUsed | language | content | length | creator | Forum.id | place
E	post_hasTag_tag_*.csv	Post.id | Tag.id

Table D.6: Files output by the CsvCompositeMergeForeign serializer (18 in total). The first part of the table contains the static entites, the second part contains the dynamic ones. Notation – C: entity category, N: node, E: edge.

- `updateStream_*_person.csv` files contain update operation 1: `INS 1`
- `updateStream_*_forum.csv` files contain update operations 2–8: `INS 2` `INS 3` `INS 4` `INS 5` `INS 6` `INS 7` `INS 8`

Remark: update streams in version 0.3.0 only contain inserts. Delete operations are being designed and will be released later.

D.1.4 Substitution Parameters

The substitution parameters are generated in the `substitution_parameters/` directory. Each parameter file is named `{interactive|bi}_<id>_param.txt`, corresponding to an operation of Interactive complex reads (`IC 1` – `IC 14`) and BI reads (`BI 1` – `BI 20`). The schemas of these files are defined by the operator, e.g. the schema of `IC 1` is "`personId|firstName`".

Warning. Note that the substitution parameter files use UNIX epoch timestamps (which should be converted to a datetime value in GMT+0).

C	File	Content
N	organisation_*.csv	id | type | name | url
E	organisation_isLocatedIn_place_*.csv	Organisation.id | Place.id
N	place_*.csv	id | name | url | type
E	place_isPartOf_place_*.csv	Place.id | Place.id
N	tag_*.csv	id | name | url
E	tag_hasType_tagclass_*.csv	Tag.id | TagClass.id
N	tagclass_*.csv	id | name | url
E	tagclass_isSubclassOf_tagclass_*.csv	TagClass.id | TagClass.id
N	comment_*.csv	creationDate | deletionDate | explicitlyDeleted | id | locationIP | browserUsed | content | length
E	comment_hasCreator_person_*.csv	creationDate | deletionDate | explicitlyDeleted | Comment.id | Person.id
E	comment_hasTag_tag_*.csv	creationDate | deletionDate | explicitlyDeleted | Comment.id | Tag.id
E	comment_isLocatedIn_place_*.csv	creationDate | deletionDate | explicitlyDeleted | Comment.id | Place.id
E	comment_replyOf_comment_*.csv	creationDate | deletionDate | explicitlyDeleted | Comment.id | ParentComment.id
E	comment_replyOf_post_*.csv	creationDate | deletionDate | explicitlyDeleted | Comment.id | ParentPost.id
N	forum_*.csv	creationDate | deletionDate | explicitlyDeleted | id | title | type
E	forum_containerOf_post_*.csv	creationDate | deletionDate | explicitlyDeleted | Forum.id | Post.id
E	forum_hasMember_person_*.csv	creationDate | deletionDate | explicitlyDeleted | Forum.id | Person.id | type
E	forum_hasModerator_person_*.csv	creationDate | deletionDate | explicitlyDeleted | Forum.id | Person.id
E	forum_hasTag_tag_*.csv	creationDate | deletionDate | explicitlyDeleted | Forum.id | Tag.id
N	person_*.csv	creationDate | deletionDate | explicitlyDeleted | id | firstName | lastName | gender | birthday | locationIP | browserUsed
A	person_email_emailaddress_*.csv	creationDate | deletionDate | explicitlyDeleted | Person.id | email
E	person_hasInterest_tag_*.csv	creationDate | deletionDate | explicitlyDeleted | Person.id | Tag.id
E	person_isLocatedIn_place_*.csv	creationDate | deletionDate | explicitlyDeleted | Person.id | Place.id
E	person_knows_person_*.csv	creationDate | deletionDate | explicitlyDeleted | Person1.id | Person2.id | weight
E	person_likes_comment_*.csv	creationDate | deletionDate | explicitlyDeleted | Person.id | Comment.id
E	person_likes_post_*.csv	creationDate | deletionDate | explicitlyDeleted | Person.id | Post.id
A	person_speaks_language_*.csv	creationDate | deletionDate | explicitlyDeleted | Person.id | language
E	person_studyAt_organisation_*.csv	creationDate | deletionDate | explicitlyDeleted | Person.id | Organisation.id | classYear
E	person_workAt_organisation_*.csv	creationDate | deletionDate | explicitlyDeleted | Person.id | Organisation.id | workFrom
N	post_*.csv	creationDate | deletionDate | explicitlyDeleted | id | imageFile | locationIP | browserUsed | language | content | length | Forum.id
E	post_hasCreator_person_*.csv	creationDate | deletionDate | explicitlyDeleted | Post.id | Person.id
E	post_hasTag_tag_*.csv	creationDate | deletionDate | explicitlyDeleted | Post.id | Tag.id
E	post_isLocatedIn_place_*.csv	creationDate | deletionDate | explicitlyDeleted | Post.id | Place.id

Table D.7: Files output by the CsvBasic serializer using "raw data" mode (33 in total). The first part of the table contains the static entites, the second part contains the dynamic ones. Notation – C: entity category, N: node, E: edge, A: attribute.

timestamp (t)	dependant timestamp (t_d)	operation id	...

Table D.8: Generic schema of update (insert) stream files.

t | t_d | 1 | personId | personFirstName | personLastName | gender | birthday | creationDate | locationIP | browserUsed | cityId | languages | emails | tagIds | studyAt | workAt
t | t_d | 2 | personId | postId | creationDate
t | t_d | 3 | personId | commentId | creationDate
t | t_d | 4 | forumId | forumTitle | creationDate | moderatorPersonId | tagIds
t | t_d | 5 | personId | forumId | creationDate
t | t_d | 6 | postId | imageFile | creationDate | locationIP | browserUsed | language | content | length | authorPersonId | forumId | countryId | tagIds
t | t_d | 7 | commentId | creationDate | locationIP | browserUsed | content | length | authorPersonId | countryId | replyToPostId | replyToCommentId | tagIds
t | t_d | 8 | person1Id | person2Id | creationDate

Table D.9: Schemas of the lines in the update (insert) stream files.

E Example graph

Figure E.1 shows a snapshot of an example graph.

Figure E.2 shows an example graph with refresh operations. Insertions are denoted with a green asterisk ✱. Deletions of a single element are denoted with a red cross ✖, while recursive deletions are denoted with a purple cross ✖.

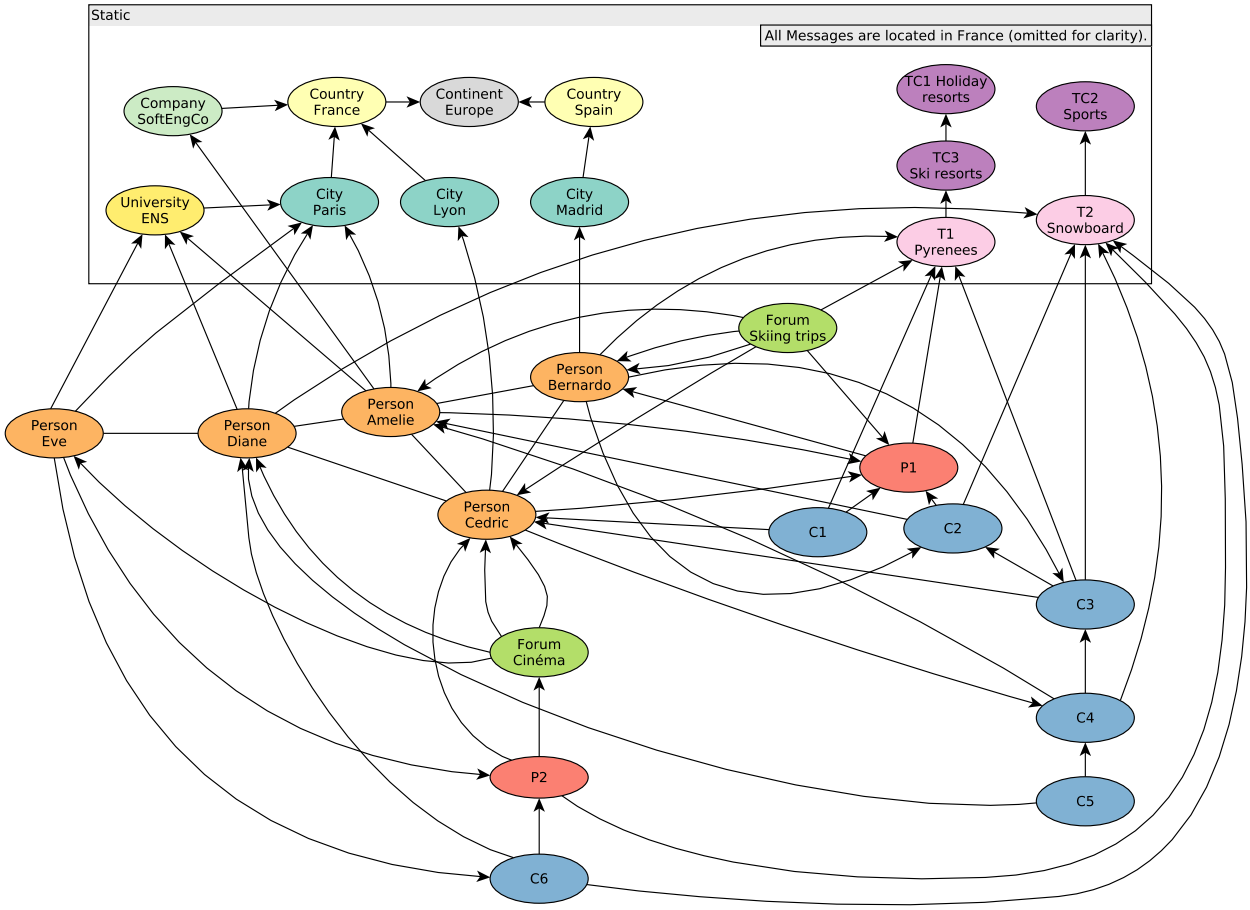


Figure E.1: Example graph snapshot (without refresh operations).

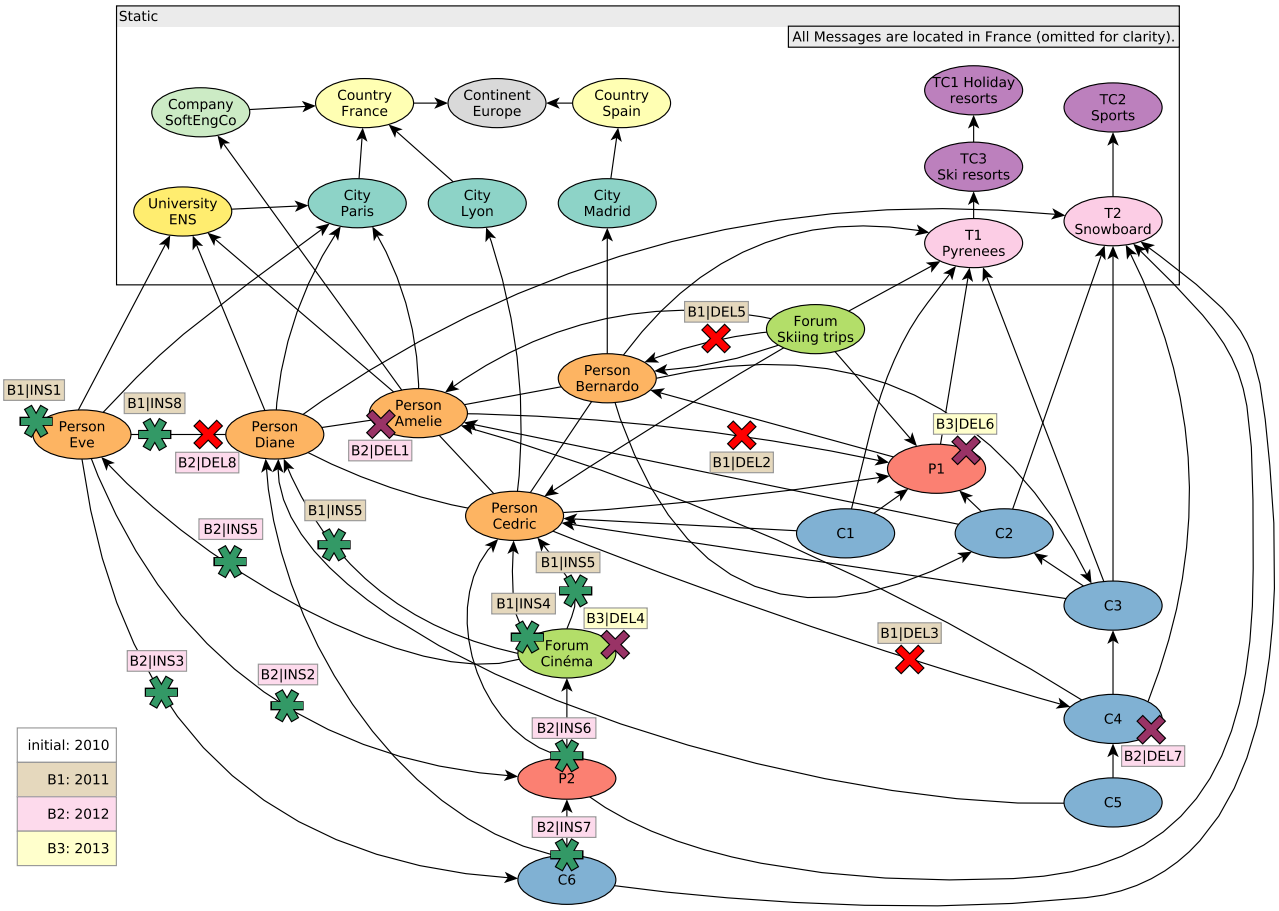


Figure E.2: Example graph with refresh operations.