LDBC

The graph & RDF benchmark reference

LDBC Social Network Benchmark (SNB) – 0.3.0

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 (Neo Technologies)
- Peter Boncz (Vrije Universiteit Amsterdam and CWI)
- Orri Erling (Google)
- Andrey Gubichev (Google)
- Moritz Kaufmann (Tableau)
- Josep Lluís Larriba Pey (Universitat Politècnica de Catalunya)
- Minh-Duc Pham (Altran)
- Marcus Paradies (SAP)
- Arnau Prat (Sparsity Technologies)
- Mirko Spasić (Openlink Software)
- Norbert Martínez (Huawei Technologies)
- Gábor Szárnyas (MTA-BME Lendület Research Group on Cyber-Physical Systems)

Definitions

DATAGEN: Is the data generator provided by the LDBC SNB, which is responsible for generating the data needed to run the benchmark.

DBMS: A DataBase Management System.

LDBC SNB: Linked Data Benchmark Council Social Network Benchmark.

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

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

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

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

Full Disclosure Report (FDR): The FDR is a document which allows reproduction of any benchmark result by a third party. This contains complete description of the SUT and the circumstances of the benchmark run, e.g. configuration of SUT, dataset and test driver, etc.

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

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

1 Introduction

1.1 Motivation for the Benchmark

The new era of data economy, based on large, distributed, and complexly structured data sets, has brought on new and complex challenges in the field of data management and analytics. These data sets, usually modeled as large graphs, have attracted both 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 [7], 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 [10], 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) 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 indexes 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.

- **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. 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 Section 2.3, 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 persons' 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 3, 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 [2] provides other benchmarks as well:

- The Semantic Publishing Benchmark (SPB)[1] measures the performance of RDF engines operating on semantic data sets.
- The Graphalytics benchmark [10] 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

[1] `http://ldbcouncil.org/benchmarks/spb`

- 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.

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. All the software provided by LDBC SNB have been developed and tested under Linux-based operating systems.

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.

2.2 Software and Useful Links

- **LDBC Driver 0.3 –** `https://github.com/ldbc/ldbc_driver`: The driver responsible for executing the LDBC SNB workload.
- **DATAGEN 0.2.5 –** `https://github.com/ldbc/ldbc_snb_datagen`: The data generator used to generate the datasets of the benchmark.

2.3 Data

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

2.3.1 Data Types

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

Type	Description
ID	integer type with 64-bit precision. All IDs within a single entity type (e.g. Post) are unique, but different entity types (e.g. a Forum and a Post) might have the same ID.
32-bit Integer	integer type with 32-bit precision
64-bit Integer	integer type with 64-bit precision
String	variable length text of size 40 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+0000*, 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 *+0000* of the end represents the timezone, which in this case is always GMT.

Table 2.1: Description of the data types.

2.3.2 Data Schema

Figure 2.1 shows the data schema in UML. The schema defines the structure of the data used in the benchmark in terms of entities and their relations. Data represents a snapshot of the activity of a social network during a period of time. Data includes entities such as Persons, 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.

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.

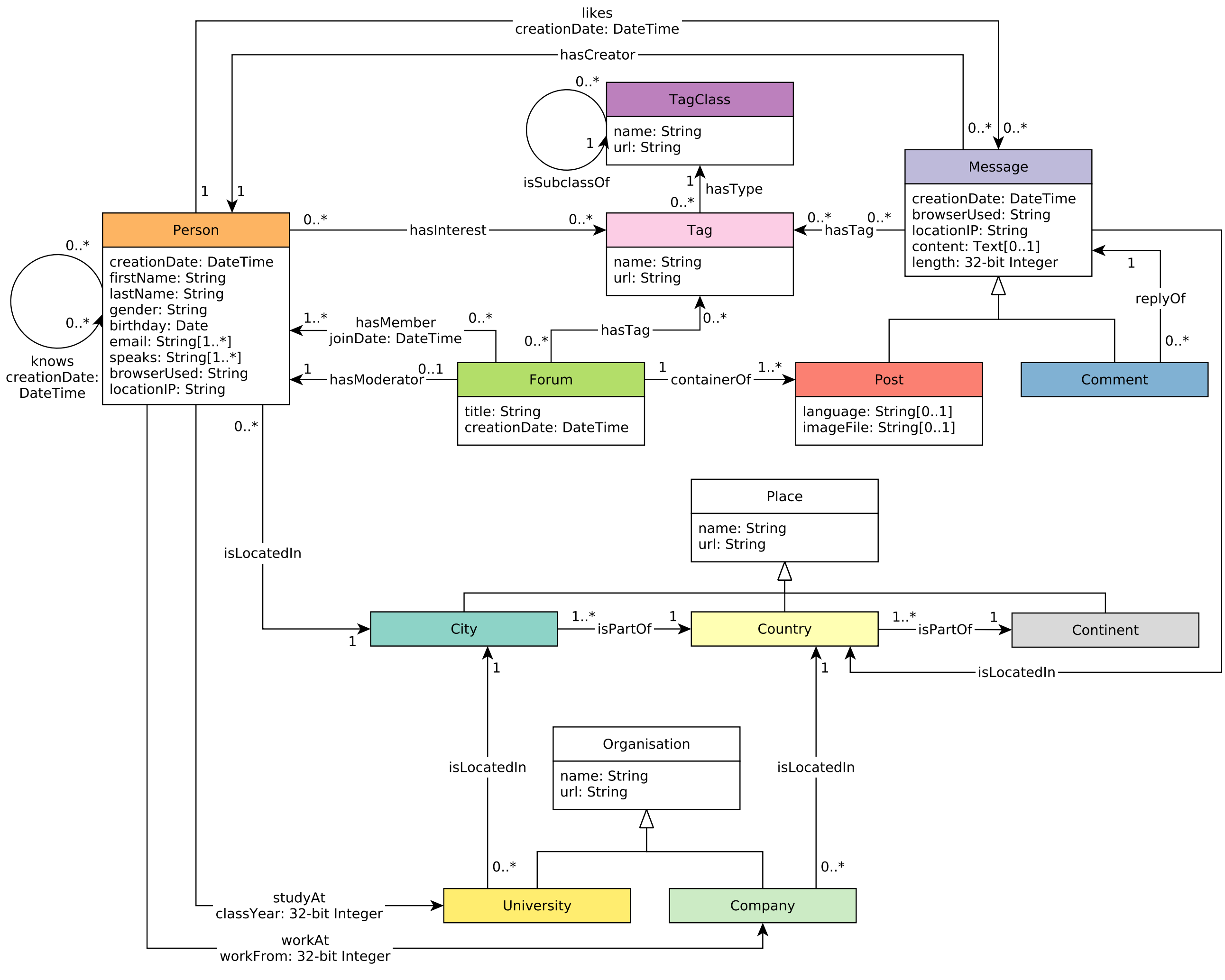


Figure 2.1: The LDBC SNB data schema

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

Textual Restrictions and Notes

- Posts have content or imageFile. They have one of them but not both. The one they do not have is an empty string.
- Posts in a forum can be created by a non-member person if and only if that person is a modeartor.

Entities

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.

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

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: persons' personal walls, image albums, and groups. They are distinguished by their titles. Table 2.2 shows the attributes of 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 2.2: Attributes of Forum entity.

Message: an abstract entity that represents a message created by a person. Table 2.3 shows the attributes of Message abstract entity.

Attribute	Type	Description
`id`	ID	The identifier of the message.
`browserUsed`	String	The browser used by the Person to create the message.
`creationDate`	DateTime	The date the message was created.
`locationIP`	String	The IP of the location from which the message was created.
`content`	Text[0..1]	The content of the message.
`length`	32-bit Integer	The length of the content.

Table 2.3: Attributes of Message interface.

Organisation: an institution of the real world. Table 2.4 shows the attributes of 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 2.4: Attributes of 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 2.5 shows the attributes of Person entity.

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

Table 2.5: Attributes of Person entity.

Place: a place in the world. Table 2.6 shows the attributes of Place entity.

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 2.6: Attributes of 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 2.7 shows the attributes of Post entity.

Attribute	Type	Description
`language`	String[0..1]	The language of the post.
`imageFile`	String[0..1]	The image file of the post.

Table 2.7: Attributes of 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 2.8 shows the attributes of 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 2.8: Attributes of Tag entity.

TagClass: a class or a category used to build a hierarchy of tags. Table 2.9 shows the attributes of 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 2.9: Attributes of TagClass entity.

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

Relations

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

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

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

Table 2.10: Description of the data relations.

Domain Concepts

A *thread* consist of Messages, starting with a single Post and Comments that transitively reply to that Post.

2.3.3 Output Data

DATAGEN produces outputs three different items:

- **Dataset**: The dataset to be bulk loaded by the SUT. 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.

The SUT have to take care only of the generated Dataset to be bulk loaded. Three different formats are supported by DATAGEN:

- **CSV:** Data output in CSV format, one file per different entity and on file per different relation. Also, there is a file for those attributes whose cardinality is larger than one (i.e. Person.email, Person.speaks).
- **CSVMergeForeign:** Similar to the CSV format, but in this case, those relations of the form 1 to 1 and 1 to N, are stored in the tail entity file as a foreign keys.
- **CSVComposite:** Similar to the CSV format, but uses composite attributes for storing the Person.email, Person.speaks attributes.
- **CSVCompositeMergeForeign:** Has the traits of both the CSVComposite and the CSVMergeForeign formats.
- **Turtle:** Dataset in Turtle format for RDF systems.

CSV

This is a comma separated format. Each entity, relation and properties with a cardinality larger than one, are output in a separate file. Generated files are summarized at Table 2.13. Depending on the number of threads used for generating the dataset, the number of files varies, since there is a file generated per thread. The * in the file names indicates a number between 0 and $\mathsf{NumberOfThreads} - 1$.

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

Table 2.11: Files output by the CSV serializer (33 in total)

CSV_MERGE_FOREIGN

This is a comma separated format. It is similar to CSV, but those relations connecting two entities A and B, where an entity A has a cardinality of one, A is output as a column of entity B. Generated files are summarized at Table 2.14. Depending on the number of threads used for generating the dataset, the number of files varies, since there is a file generated per thread. The * in the file names indicates a number between 0 and $\mathsf{NumberOfThreads} - 1$.

Turtle

This is the standard Turtle[1] format. DATAGEN outputs two files: `0_ldbc_socialnet_static_dbp.ttl` and `0_ldbc_socialnet.ttl`.

2.3.4 Scale Factors

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

The size of the resulting dataset, is mainly affected by the following configuration parameters: the number of persons and the number of years simulated. Different SFs are computed by scaling the number of Persons in

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

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

Table 2.12: Files output by the CSV_MERGE_FOREIGN serializer (20 in total)

the network, while fixing the number of years simulated. Table 2.15 shows the parameters used in each of the SFs.

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

Table 2.15: Parameters of each scale factor.

For example, SF 30 consists of the activity of a social network of 182K users during a period of three years, starting from 2010.

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

Table 2.13: Files output by the CSV_COMPOSITE serializer (31 in total)

File	Content
comment_*.csv	id | creationDate | locationIP | browserUsed | content | length | creator | place | replyOfPost | replyOfComment
comment_hasTag_tag_*.csv	Comment.id | Tag.id
forum_*.csv	id | title | creationDate | moderator
forum_hasMember_person_*.csv	Forum.id | Person.id | joinDate
forum_hasTag_tag_*.csv	Forum.id | Tag.id
organisation_*.csv	id | type("university", "company") | name | url
person_*.csv	id | firstName | lastName | gender | birthday | creationDate | locationIP | browserUsed | place | language | emails
person_hasInterest_tag_*.csv	Person.id | Tag.id
person_knows_person_*.csv	Person.id | Person.id | creationDate
person_likes_comment_*.csv	Person.id | Post.id | creationDate
person_likes_post_*.csv	Person.id | Post.id | creationDate
person_studyAt_organisation_*.csv	Person.id | Organisation.id | classYear
person_workAt_organisation_*.csv	Person.id | Organisation.id | workFrom
place_*.csv	id | name | url | type("city", "country", "continent")
post_*.csv	id | imageFile | creationDate | locationIP | browserUsed | language | content | length | creator | Forum.id | place
post_hasTag_tag_*.csv	Post.id | Tag.id
tag_*.csv	id | name | url | hasType
tagclass_*.csv	id | name | url | isSubclassOf

Table 2.14: Files output by the CSV_COMPOSITE_MERGE_FOREIGN serializer (18 in total)

3 Workloads

3.1 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; and *results*, a list of expected results and their types. The syntax used in *parameters* and *results* sections is as follows:

- **Entity**: entity type in the dataset.
  One word, possibly constructed by appending multiple words together, starting with uppercase character and following the camel case notation, e.g. `TagClass` represents an entity of type "TagClass".
- **Relationship**: relationship type in the dataset.
  One word, possibly constructed by appending multiple words together, starting with lowercase character and following the camel case notation, and surrounded by arrow to communicate direction, e.g. `-worksAt->` represents a directed relationship of type "worksAt".
- **Attribute**: attribute of an entity or relationship in the dataset.
  One word, possibly constructed by appending multiple words together, starting with lowercase character and following the camel case notation, and prefixed by a "." to dereference the entity/relationship, e.g. `Person.firstName` refers to "firstName" attribute on the "Person" entity, and `-studyAt->.classYear` refers to "classYear" attribute on the "studyAt" relationship.
- **Unordered Set**: an unordered collection of distinct elements.
  Surrounded by { and } braces, with the element type between them, e.g. `{String}` refers to a set of strings.
- **Ordered List**: an 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 is returned with an unmodified value and type.
- **Calculated** (`C`), if the result is calculated from other values and conditions.
- **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. $\mathsf{count}(x) + \mathsf{count}(y)$ or $\mathsf{avg}(x + y)$), it is considered an aggregated result.
- **Meta** (`M`), if the result is based on type information, e.g. the type of the node.

3.2 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)` denoted 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` , `BI 2` , etc.) 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+0000 (i.e. with the timezone of GMT).

Matching semantics. Unless noted otherwise, the specification uses *homomorphic* matching semantics [1], 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 3.1 with the following notation:

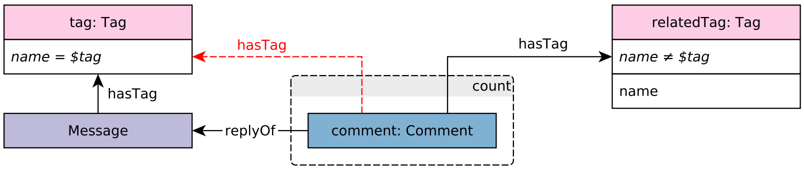


Figure 3.1: Example graph pattern.

- Nodes are marked as 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 not used in the query results, aggregations or calculations, and not referenced in the query specification, the entityName can be omitted.
- Positive conditions for edges are denoted with solid lines.
- Negative conditions for edges, i.e. edges that are not allowed in the graph, are denoted with dashed red lines.
- Edges without direction imply that there must be an edge in *at least one of the directions*.
- 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 $* 1 \ldots 2$. By default, the value of min is 1, and the value of max is unlimited.
- Aggregations are shown in dashed boxes with the type of aggregation (count, sum, avg, etc.) in the upper right corner.

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

- UNWIND unnests a list, i.e. produces a set of one-tuples. For example, UNWIND$[1, 2, 3]$ results in $\{\langle 1\rangle, \langle 2\rangle, \langle 3\rangle\}$.
- Aggregation operations: `count`, `avg`.
- 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).

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*.

3.3 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 updates.

3.4 Load Definition

LDBC SNB 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 3.1 shows the frequencies assigned to each query type for SF1. The frequencies of the different scale factors can be found in Section B.1.

Query Type	freq	Query Type	freq
Query 1	26	Query 8	45
Query 2	37	Query 9	157
Query 3	69	Query 10	30
Query 4	36	Query 11	16
Query 5	57	Query 12	44
Query 6	129	Query 13	19
Query 7	87	Query 14	49

Table 3.1: Frequencies for each query type for SF1.

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 3.1). Therefore, different throughputs can be tested while maintaining the relative ratios between the different query types.

4 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 neighborhood 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 4.1.
- **Short read-only queries.** See Section 4.2.
- **Transactional update queries.** See Section 4.3.

A detailed description of the workload is available in the 2015 SIGMOD paper [7].

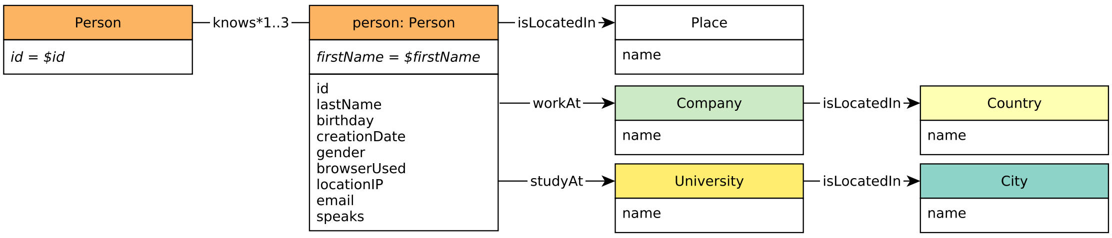
IC 1
IC 2
IC 3
IC 4
IC 5
IC 6
IC 7
IC 8
IC 9
IC 10
IC 11
IC 12
IC 13
IC 14

4.1 Complex Reads

Interactive / complex / 1

query	Interactive / complex / 1
title	Friends with certain name

pattern



desc.	Given a start Person, find Persons with a given first name that the start Person is connected to (excluding start Person) by at most 3 steps via Knows relationships. Return Persons, including the distance (1..3), summaries of the Persons workplaces and places of study.

params

1	Person.id	ID	Person
2	Person.firstName	String	Name

result

1	Person.id	ID	R	friendId
2	Person.lastName	String	R	friendLastName
3	distanceFromPerson	32-bit Integer	R	distanceFromPerson
4	Person.birthday	Date	R	friendBirthday
5	Person.creationDate	DateTime	R	friendCreationDate
6	Person.gender	String	R	friendGender
7	Person.browserUsed	String	R	friendBrowserUsed
8	Person.locationIP	String	R	friendLocationIp
9	{Person.email}	{String}	R	friendEmails
10	{Person.speaks}	{String}	R	friendLanguages
11	Person-isLocatedIn->Place.name	String	R	friendCityName
12	{Person-studyAt->University.name, Person-studyAt->.classYear, Person-studyAt->University-isLocatedIn->City.name}	{<String, 32-bit Integer, String>}	R	friendUniversities
13	{Person-workAt->Company.name, Person-workAt->.workFrom, Person-workAt->Company-isLocatedIn->Country.name}	{<String, 32-bit Integer, String>}	R	friendCompanies

sort

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

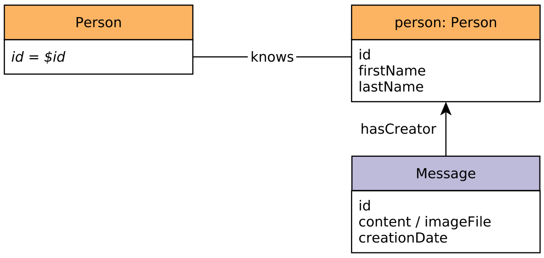
limit	20
CPs	2.1, 5.3
relevance	This query is a representative of a simple navigational query. It looks for paths of length 1..3 through the knows relation, starting from a given Person and ending at a Person with a given first name. It is interesting for several aspects. (1) It requires for a complex aggregation for returning the c ncatenation 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.

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

Interactive / complex / 2

query	Interactive / complex / 2
title	Recent posts and comments by your friends

pattern



desc.	Given a start Person, find (most recent) Messages from all of that Person's friends, that were created before (and including) a given date.

params

1	Person.id	ID	Person
2	date	DateTime	Date0

result

1	Message-hasCreator->Person.id	ID	R	personId
2	Message-hasCreator->Person.firstName	String	R	personFirstName
3	Message-hasCreator->Person.lastName	String	R	personLastName
4	Message.id	ID	R	postOrCommentId
5	Message.content or Post.imageFile	String	R	postOrCommentContent
6	Message.creationDate	DateTime	R	postOrCommentCreationDate

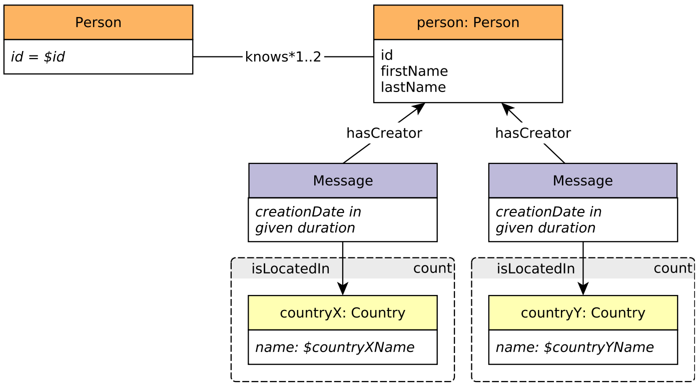
sort

1	Message.creationDate	↓
2	Message.id	↑

limit	20
CPs	1.1, 2.2, 2.3, 3.2
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.

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

Interactive / complex / 3

query	Interactive / complex / 3
title	Friends and friends of friends that have been to countries X and Y
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, X and Y, within a given period. Only Persons that are foreign to Countries X and Y are considered, that is Persons whose Location is not Country X or Country Y.

params

1	`Person.id`	ID	`Person`
2	`CountryX.name`	String	`Country1`
3	`CountryY.name`	String	`Country2`
4	`startDate`	Date	`Date0` – Beginning of requested period
5	`duration`	32-bit Integer	`Duration` – Duration of requested period, in days the interval [startDate, startDate + Duration) is closed-open

result

1	`Person.id`	ID	R	`personId`
2	`Person.firstName`	String	R	`personFirstName`
3	`Person.lastName`	String	R	`personLastName`
4	`xCount`	32-bit Integer	A	`xCount` – Number of Messages from Country X made by Person within the given time
5	`yCount`	32-bit Integer	A	`yCount` – Number of Messages from Country Y made by Person within the given time
6	`count`	32-bit Integer	A	`count = xCount + yCount`

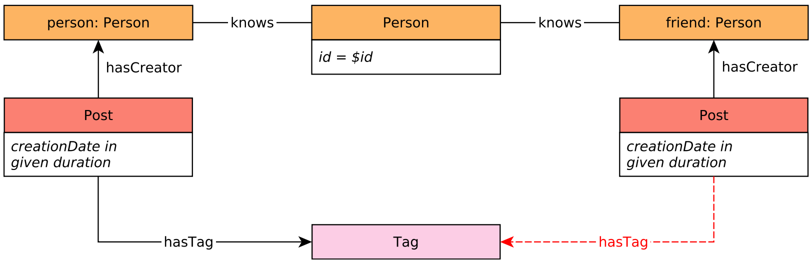
sort

1	`xCount`	↓
2	`Person.id`	↑

limit	20
CPs	2.1, 3.1, 5.1
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 X and Y, 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 X and Y.

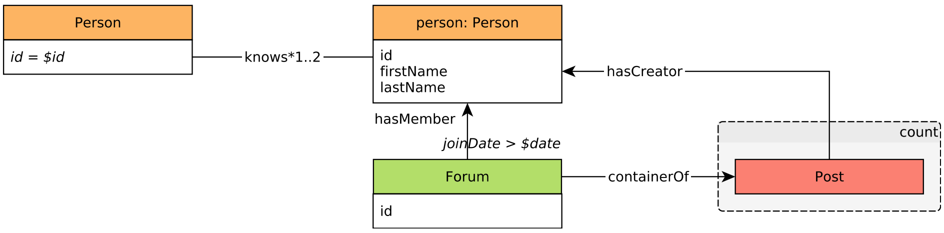
Interactive / complex / 4

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

<table>
<tr><td>query</td><td>Interactive / complex / 4</td></tr>
<tr><td>title</td><td>New topics</td></tr>
<tr><td>pattern</td><td>
</td></tr>
<tr><td>desc.</td><td>Given a start Person, find Tags that are attached to Posts that were created by that Person's friends. Only include Tags that were attached to friends' Posts created within a given time interval, and that were never attached to friends' Posts created before this interval.</td></tr>
<tr><td>params</td><td><table>
<tr><td>1</td><td>Person.id</td><td>ID</td><td>Person</td></tr>
<tr><td>2</td><td>startDate</td><td>Date</td><td>Date0</td></tr>
<tr><td>3</td><td>duration</td><td>32-bit Integer</td><td>Duration – Duration of requested period, in days. The interval [startDate, startDate + Duration) is closed-open</td></tr>
</table></td></tr>
<tr><td>result</td><td><table>
<tr><td>1</td><td>Tag.name</td><td>String</td><td>R</td><td>tagName</td></tr>
<tr><td>2</td><td>postCount</td><td>32-bit Integer</td><td>A</td><td>postCount – Number of Posts made within the given time interval that have this Tag</td></tr>
</table></td></tr>
<tr><td>sort</td><td><table>
<tr><td>1</td><td>postCount</td><td>↓</td><td></td></tr>
<tr><td>2</td><td>Tag.name</td><td>↑</td><td></td></tr>
</table></td></tr>
<tr><td>limit</td><td>10</td></tr>
<tr><td>CPs</td><td>2.3</td></tr>
<tr><td>relevance</td><td>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.</td></tr>
</table>

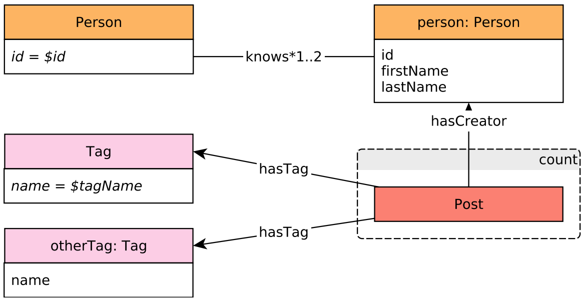
IC 1
IC 2
IC 3
IC 4
IC 5
IC 6
IC 7
IC 8
IC 9
IC 10
IC 11
IC 12
IC 13
IC 14

Interactive / complex / 5

<table>
<tr><td>query</td><td>Interactive / complex / 5</td></tr>
<tr><td>title</td><td>New groups</td></tr>
<tr><td>pattern</td><td>
</td></tr>
<tr><td>desc.</td><td>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.</td></tr>
<tr><td>params</td><td><table>
<tr><td>1</td><td>Person.id</td><td>ID</td><td>Person</td></tr>
<tr><td>2</td><td>date</td><td>Date</td><td>Date0</td></tr>
</table></td></tr>
<tr><td>result</td><td><table>
<tr><td>1</td><td>Forum.title</td><td>String</td><td>R</td><td>forumTitle</td></tr>
<tr><td>2</td><td>postCount</td><td>32-bit Integer</td><td>A</td><td>postCount – Number of Posts made in Forum that were created by friends</td></tr>
</table></td></tr>
<tr><td>sort</td><td><table>
<tr><td>1</td><td>postCount</td><td>↓</td><td></td></tr>
<tr><td>2</td><td>Forum.id</td><td>↑</td><td></td></tr>
</table></td></tr>
<tr><td>limit</td><td>20</td></tr>
<tr><td>CPs</td><td>2.3, 3.3</td></tr>
<tr><td>relevance</td><td>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 indexes, 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 indexes will be highly beneficial.</td></tr>
</table>

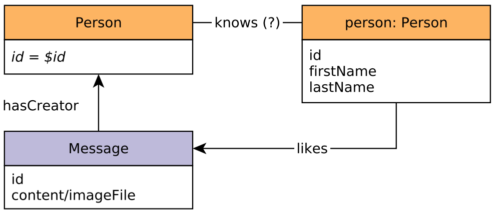
Interactive / complex / 6

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 For each Tag, find the count of Posts that were created by these Persons, which contain both this Tag and the given Tag.
params	1 · Person.id · ID · Person 2 · Tag.name · String · Tag
result	1 · Tag.name · String · R · tagName 2 · postCount · 32-bit Integer · A · postCount – Number of Posts that were created by friends and friends of friends, which contain this Tag
sort	1 · postCount · ↓ 2 · Tag.name · ↑
limit	10
CPs	5.1
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 (most recent) Likes on any of start Person's Messages. Find Persons that Liked any of start Person's Messages, the Messages they liked most recently, creation date of that Like, and the latency (in minutes) between creation of Messages and Like. Additionally, for each Person found return a flag indicating whether the liker is a friend of start Person. In the case that a Person Liked multiple Messages at the same time, return the Message with lowest identifier.

params

1	`Person.id`	64-bit Integer	`Person`

result

1	`Person.id`	ID	R	`personId`
2	`Person.firstName`	String	R	`personFirstName`
3	`Person.lastName`	String	R	`personLastName`
4	`Like.creationDate`	DateTime	R	`likeCreationDate`
5	`Message.id`	ID	R	`commentOrPostId`
6	`Message.content or Post.imageFile`	String	R	`commentOrPostContent`
7	`minutesLatency`	32-bit Integer	C	`minutesLatency` – Duration between creation of Message and Like, in minutes
8	`isNew`	Boolean	C	`isNew` – `false` if liker Person is friend of start Person, `true` otherwise

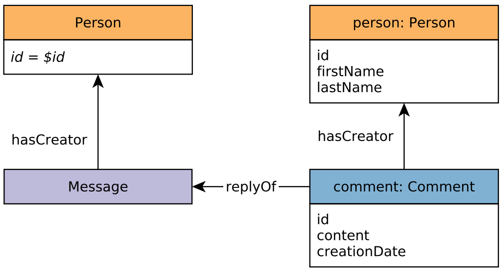
sort

1	`Like.creationDate`	↓
2	`Person.id`	↑

limit	20
CPs	2.2, 2.3, 3.3, 5.1
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 2-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.

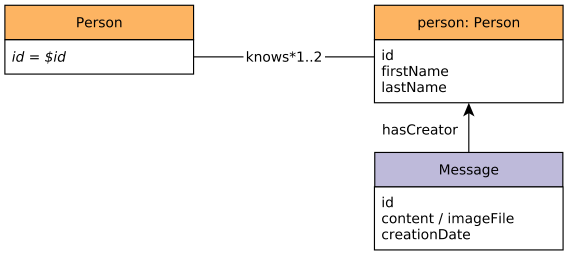
IC 1
IC 2
IC 3
IC 4
IC 5
IC 6
IC 7
IC 8
IC 9
IC 10
IC 11
IC 12
IC 13
IC 14

Interactive / complex / 8

<table>
<tr><td>query</td><td>Interactive / complex / 8</td></tr>
<tr><td>title</td><td>Recent replies</td></tr>
<tr><td>pattern</td><td>
</td></tr>
<tr><td>desc.</td><td>Given a start Person, find (most recent) Comments that are replies to Messages of the start Person. Only consider immediate (1-hop) replies, not the transitive (multi-hop) case. Return the reply Comments, and the Person that created each reply Comment.</td></tr>
<tr><td>params</td><td><table>
<tr><td>1</td><td>Person.id</td><td>ID</td><td>Person</td></tr>
</table></td></tr>
<tr><td>result</td><td><table>
<tr><td>1</td><td>Person.id</td><td>ID</td><td>R</td><td>personId</td></tr>
<tr><td>2</td><td>Person.firstName</td><td>String</td><td>R</td><td>personFirstName</td></tr>
<tr><td>3</td><td>Person.lastName</td><td>String</td><td>R</td><td>personLastName</td></tr>
<tr><td>4</td><td>Comment.creationDate</td><td>DateTime</td><td>R</td><td>commentCreationDate</td></tr>
<tr><td>5</td><td>Comment.id</td><td>ID</td><td>R</td><td>commentId</td></tr>
<tr><td>6</td><td>Comment.content</td><td>String</td><td>R</td><td>commentContent</td></tr>
</table></td></tr>
<tr><td>sort</td><td><table>
<tr><td>1</td><td>Comment.creationDate</td><td>↓</td><td></td></tr>
<tr><td>2</td><td>Comment.id</td><td>↑</td><td></td></tr>
</table></td></tr>
<tr><td>limit</td><td>20</td></tr>
<tr><td>CPs</td><td>2.4, 3.2, 3.3, 5.3</td></tr>
<tr><td>relevance</td><td>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.</td></tr>
</table>

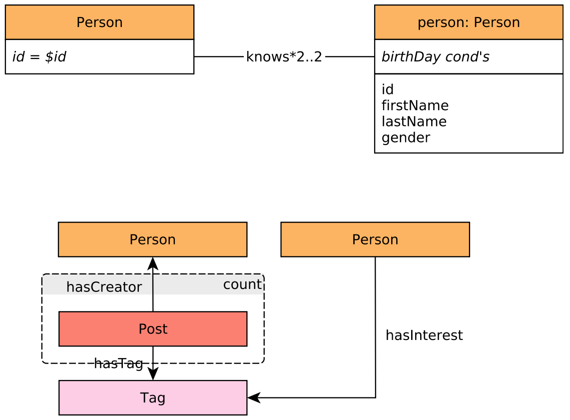
IC 1
IC 2
IC 3
IC 4
IC 5
IC 6
IC 7
IC 8
IC 9
IC 10
IC 11
IC 12
IC 13
IC 14

Interactive / complex / 9

<table>
<tr><td>query</td><td>Interactive / complex / 9</td></tr>
<tr><td>title</td><td>Recent posts and comments by friends or friends of friends</td></tr>
<tr><td>pattern</td><td>
</td></tr>
<tr><td>desc.</td><td>Given a start Person, find the (most recent) Messages created by that Person's friends or friends of friends (excluding start Person). Only consider the Messages created before a given date (excluding that date).</td></tr>
<tr><td>params</td><td><table>
<tr><td>1</td><td>Person.id</td><td>ID</td><td>Person</td></tr>
<tr><td>2</td><td>date</td><td>Date</td><td>Date0</td></tr>
</table></td></tr>
<tr><td>result</td><td><table>
<tr><td>1</td><td>Message-hasCreator->Person.id</td><td>ID</td><td>R</td><td>personId</td></tr>
<tr><td>2</td><td>Message-hasCreator->Person.firstName</td><td>String</td><td>R</td><td>personFirstName</td></tr>
<tr><td>3</td><td>Message-hasCreator->Person.lastName</td><td>String</td><td>R</td><td>personLastName</td></tr>
<tr><td>4</td><td>Message.id</td><td>ID</td><td>R</td><td>commentOrPostId</td></tr>
<tr><td>5</td><td>Message.content or Post.imageFile</td><td>String</td><td>R</td><td>commentOrPostContent</td></tr>
<tr><td>6</td><td>Message.creationDate</td><td>DateTime</td><td>R</td><td>commentOrPostCreationDate</td></tr>
</table></td></tr>
<tr><td>sort</td><td><table>
<tr><td>1</td><td>Message.creationDate</td><td>↓</td><td></td></tr>
<tr><td>2</td><td>Message.id</td><td>↑</td><td></td></tr>
</table></td></tr>
<tr><td>limit</td><td>20</td></tr>
<tr><td>CPs</td><td>1.1, 1.2, 2.2, 2.3, 3.2, 3.3</td></tr>
<tr><td>relevance</td><td>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.</td></tr>
</table>

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

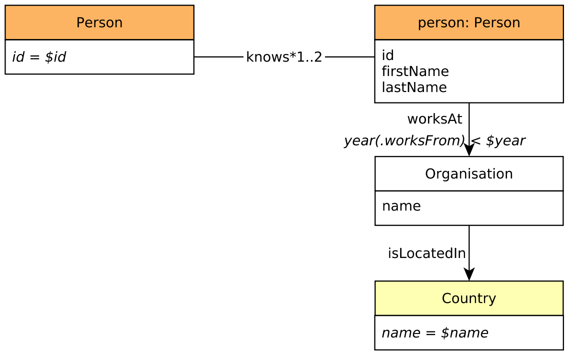
Interactive / complex / 10

<table>
<tr><td>query</td><td>Interactive / complex / 10</td></tr>
<tr><td>title</td><td>Friend recommendation</td></tr>
<tr><td>pattern</td><td>


</td></tr>
<tr><td>desc.</td><td>Given a start Person, find that Person's friends of friends (excluding start Person, and immediate friends), who were born on or after the 21st of a given month (in any year) and before the 22nd of the following month. Calculate the similarity between each of these Persons and start Person, where commonInterestScore is defined as follows: number of Posts created by that Person, such that the Post has a Tag that start Person is Interested in</td></tr>
<tr><td>params</td><td>
<table>
<tr><td>1</td><td>Person.id</td><td>ID</td><td>Person</td></tr>
<tr><td>2</td><td>month</td><td>32-bit Integer</td><td>HS0 – Between 1-12</td></tr>
</table>
</td></tr>
<tr><td>result</td><td>
<table>
<tr><td>1</td><td>Person.id</td><td>ID</td><td>R</td><td>personId</td></tr>
<tr><td>2</td><td>Person.firstName</td><td>String</td><td>R</td><td>personFirstName</td></tr>
<tr><td>3</td><td>Person.lastName</td><td>String</td><td>R</td><td>personLastName</td></tr>
<tr><td>4</td><td>commonInterestScore</td><td>32-bit Integer</td><td>C</td><td>commonInterestScore</td></tr>
<tr><td>5</td><td>Person.gender</td><td>String</td><td>R</td><td>personGender</td></tr>
<tr><td>6</td><td>Person-isLocatedIn->Place.name</td><td>String</td><td>R</td><td>personCityName</td></tr>
</table>
</td></tr>
<tr><td>sort</td><td>
<table>
<tr><td>1</td><td>similarity</td><td>↓</td><td></td></tr>
<tr><td>2</td><td>Person.id</td><td>↑</td><td></td></tr>
</table>
</td></tr>
<tr><td>limit</td><td>10</td></tr>
<tr><td>CPs</td><td>2.3, 3.3, 4.1, 4.2, 5.1, 5.2, 6.1, 7.1</td></tr>
<tr><td>relevance</td><td>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.</td></tr>
</table>

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	Person.id	ID	Person
2	Country.name	String	Country
3	year	32-bit Integer	Date0

result

1	Person.id	ID	R	personId
2	Person.firstName	String	R	personFirstName
3	Person.lastName	String	R	personLastName
4	Person-worksAt->Organisation.name	String	R	organizationName
5	Person-worksAt->.worksFrom	32-bit Integer	R	organizationWorkFromYear

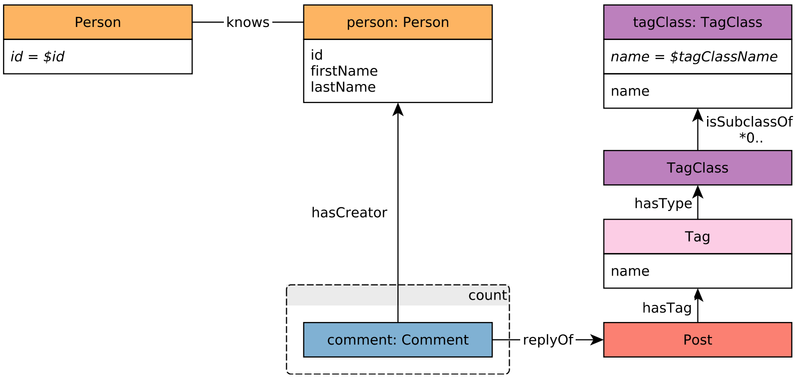
sort

1	Person-worksAt->.worksFrom	↑	
2	Person.id	↑	
3	Person-worksAt->Organisation.name	↓	

limit	10
CPs	1.3, 2.3, 2.4, 3.3
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.

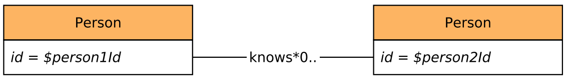
Interactive / complex / 12

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

<table>
<tr><td>query</td><td>Interactive / complex / 12</td></tr>
<tr><td>title</td><td>Expert search</td></tr>
<tr><td>pattern</td><td>
</td></tr>
<tr><td>desc.</td><td>Given a start Person, find the Comments that this Person's friends made in reply to Posts, considering only those Comments that are immediate (1-hop) replies to Posts, not the transitive (multi-hop) case. 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.</td></tr>
<tr><td>params</td><td><table>
<tr><td>1</td><td>Person.id</td><td>ID</td><td>Person</td></tr>
<tr><td>2</td><td>TagClass.name</td><td>String</td><td>TagType</td></tr>
</table></td></tr>
<tr><td>result</td><td><table>
<tr><td>1</td><td>Person.id</td><td>ID</td><td>R</td><td>personId</td></tr>
<tr><td>2</td><td>Person.firstName</td><td>String</td><td>R</td><td>personFirstName</td></tr>
<tr><td>3</td><td>Person.lastName</td><td>String</td><td>R</td><td>personLastName</td></tr>
<tr><td>4</td><td>{Tag.name}</td><td>{String}</td><td>R</td><td>tagNames</td></tr>
<tr><td>5</td><td>replyCount</td><td>32-bit Integer</td><td>A</td><td>replyCount – Number of reply Comments</td></tr>
</table></td></tr>
<tr><td>sort</td><td><table>
<tr><td>1</td><td>count</td><td>↓</td><td></td></tr>
<tr><td>2</td><td>Person.id</td><td>↑</td><td></td></tr>
</table></td></tr>
<tr><td>limit</td><td>20</td></tr>
<tr><td>CPs</td><td>3.3, 7.2, 7.3</td></tr>
<tr><td>relevance</td><td>This query looks for paths of length three, starting at a Person, moving to its friends, the to their Comments and ending at the Post the Comments are replying. The chain from original post to the reply is transitive. The traversal may be initiated at either end, the system may note that this is a tree, hence leaf to root is always best. Additionally, a hash table can be built from either end, e.g. from the friends of self, from the tags in the category, from the or other.</td></tr>
</table>

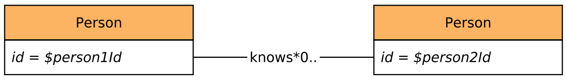
IC 1
IC 2
IC 3
IC 4
IC 5
IC 6
IC 7
IC 8
IC 9
IC 10
IC 11
IC 12
IC 13
IC 14

Interactive / complex / 13

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: regular case
params	1 | `person1.id` | ID | `Person1` 2 | `person2.id` | ID | `Person2`
result	1 | `length` | 32-bit Integer | C | `shortestPathLength`
CPs	3.3, 7.2, 7.3
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 prunning, will be crucial. This query also allows for possible parallel implementations.

Interactive / complex / 14

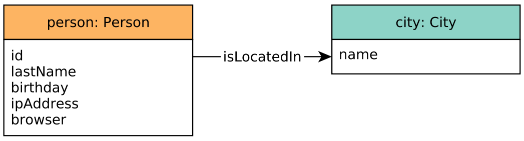
IC 1
IC 2
IC 3
IC 4
IC 5
IC 6
IC 7
IC 8
IC 9
IC 10
IC 11
IC 12
IC 13
IC 14

query	Interactive / complex / 14
title	Weighted/unweighted 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 such that every reply (by one of the Persons) to a Post (by the other Person) contributes 1.0, and every reply (by ones of the Persons) to a Comment (by the other Person) contributes 0.5. Return all the paths with shortest length, and their weights. Do not return any rows if there is now path between the two Persons.
params	1 | `person1.id` | ID | `Person1` 2 | `person2.id` | ID | `Person2`
result	1 | `[Person.id]` | [ID] | C | `personIdsInPath` – identifiers representing an ordered sequence of the Persons in the path 2 | `weight` | 64-bit Float | C | `pathWeight`
sort	1 | `weight` | ↓ | The order of paths with the same weight is unspecified
CPs	3.3, 7.2, 7.3
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 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.

4.2 Short Reads

Interactive / short / 1

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

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

params

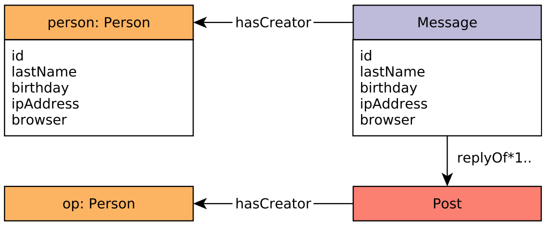
1	`Person.id`	ID

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	`Person-isLocatedIn->Place.id`	32-bit Integer	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	Person Recent Messages
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, and the author of that post. If any of the Messages is a Post, then the original Post will be the same Message, i.e. that Message will appear twice in that result.
params	1 | `Person.id` | ID
result	see below
sort	see below

params

1	`Person.id`	ID

result

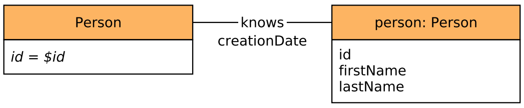
1	`Message.id`	64-bit Integer	R
2	`Message.content or Post.imageFile`	String	R
3	`Message.creationDate`	DateTime	R
4	`Post.id or Comment-replyOf*->Post.id`	ID	R
5	`Post-hasCreator->Person.id or Comment-replyOf*->Post-hasCreator->Person.id`	ID	R
6	`Post-hasCreator->Person.firstName or Comment-replyOf*->Post-hasCreator->Person.firstName`	String	R
7	`Post-hasCreator->Person.lastName or Comment-replyOf*->Post-hasCreator->Person.lastName`	String	R

sort

1	`Message.creationDate`	↓
2	`Message.id`	↓

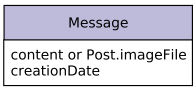
Interactive / short / 3

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

query	Interactive / short / 3
title	Person Friends
pattern	
desc.	Given a start Person, retrieve all of their friends, and the date at which they became friends.
params	1 Person.id ID
result	1 Person.id ID R 2 Person.firstName String R 3 Person.lastName String R 4 Knows.creationDate DateTime R
sort	1 Knows.creationDate ↓ 2 Person.id ↑

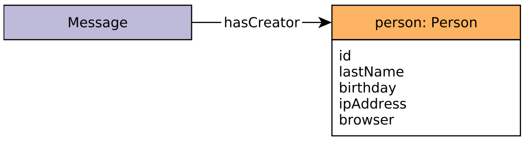
Interactive / short / 4

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

query	Interactive / short / 4
title	Message Content
pattern	
desc.	Given a Message, retrieve its content and creation date.
params	1 Message.id ID
result	1 Message.creationDate ID R 2 Message.content or Post.imageFile String R

Interactive / short / 5

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

query	Interactive / short / 5
title	Message Creator
pattern	
desc.	Given a Message, retrieve its author.

params

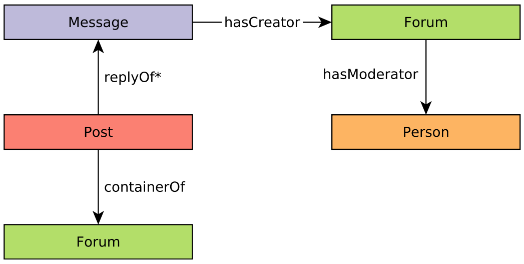
1	`Message.id`	ID

result

1	`Message-hasCreator->Person.id`	ID	R
2	`Message-hasCreator->Person.firstName`	String	R
3	`Message-hasCreator->Person.lastName`	String	R

Interactive / short / 6

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

query	Interactive / short / 6
title	Message Forum
pattern	
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.

params

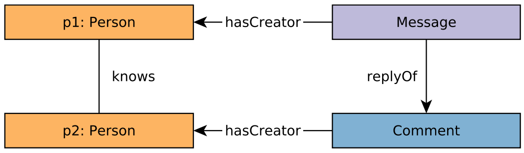
1	`Message.id`	ID

result

1	`Message<-containerOf-Forum.id`	ID	R
2	`Message<-containerOf-Forum.title`	String	R
3	`Message<-containerOf-Forum-hasModerator->Person.id`	ID	R
4	`Message<-containerOf-Forum-hasModerator->Person.firstName`	String	R
5	`Message<-containerOf-Forum-hasModerator->Person.lastName`	String	R

Interactive / short / 7

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

query	Interactive / short / 7
title	Message Replies
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 knows the author of the original message. If author is same as original author, return false for `knows` flag.
params	1 | `Message.id` | ID

result

1	`Message<-replyOf-Comment.id`	ID	R	
2	`Message<-replyOf-Comment.content`	String	R	
3	`Message<-replyOf-Comment.creationDate`	DateTime	R	
4	`Comment-hasCreator->Person.id`	ID	R	
5	`Comment-hasCreator->Person.firstName`	String	R	
6	`Comment-hasCreator->Person.lastName`	String	R	
7	`knows`	Boolean	C	Original message author knows reply author

sort

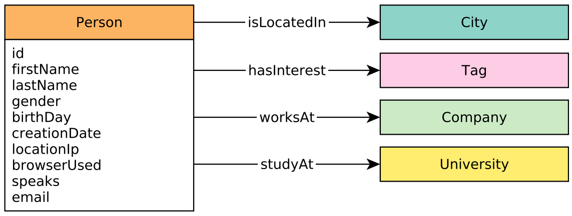
1	`Message<-replyOf-Comment.creationDate`	↓
2	`Message-hasCreator->Person.id`	↑

4.3 Updates

Each update query inserts either (1) a single node of a certain type, along with its edges to other existing nodes or (2) a single edge of a certain type between two existing nodes.

Interactive / update / 1

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

query	Interactive / update / 1
title	Add Person
pattern	
desc.	Add a Person **node**, connected to the network by 4 possible **edge** types.
params	see below

params:

1	`Person.id`	ID	
2	`Person.firstName`	String	
3	`Person.lastName`	String	
4	`Person.gender`	String	
5	`Person.birthDay`	Date	
6	`Person.creationDate`	DateTime	
7	`Person.locationIp`	String	
8	`Person.browserUsed`	String	
9	`Person-isLocatedIn->City.id`	ID	
10	`Person.speaks`	{String}	
11	`Person.email`	{String}	
12	`Person-hasInterest->Tag.id`	{ID}	
13	`(Person-studyAt->University.id, Person-studyAt->.classYear)`	{(ID, 32-bit Integer)}	
14	`(Person-workAt->Company.id, Person-workAt->.workFrom)`	{(ID, 32-bit Integer)}	

Interactive / update / 2

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

query	Interactive / update / 2
title	Add Post Like
pattern	
desc.	Add a likes **edge** to a Post.
params	see below

params:

1	`Person.id`	ID	
2	`Post.id`	ID	
3	`Person-likes->.creationDate`	DateTime	

Interactive / update / 3

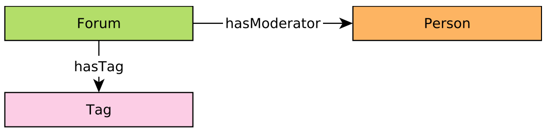
IU 1 IU 2 IU 3 IU 4 IU 5 IU 6 IU 7 IU 8

query	Interactive / update / 3
title	Add Comment Like
pattern	
desc.	Add a likes **edge** to a Comment.
params	1 Person.id — ID 2 Comment.id — ID 3 Person-likes->.creationDate — DateTime

#	Parameter	Type	
1	Person.id	ID	
2	Comment.id	ID	
3	Person-likes->.creationDate	DateTime	

Interactive / update / 4

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

query	Interactive / update / 4
title	Add Forum
pattern	
desc.	Add a Forum **node**, connected to the network by 2 possible **edge** types.
params	see below

#	Parameter	Type	
1	Forum.id	ID	
2	Forum.title	String	
3	Forum.creationDate	DateTime	
4	Forum-hasModerator->Person.id	ID	
5	Forum-hasTag->Tag.id	{ID}	

Interactive / update / 5

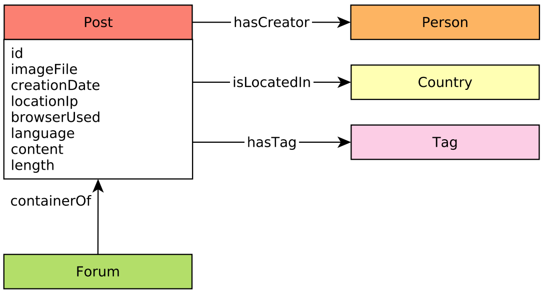
IU 1 IU 2 IU 3 IU 4 IU 5 IU 6 IU 7 IU 8

query	Interactive / update / 5
title	Add Forum Membership
pattern	
desc.	Add a Forum membership **edge** (hasMember) to a Person.
params	see below

#	Parameter	Type	
1	Person.id	ID	
2	Forum-hasMember->Person.id	ID	
3	Forum-hasMember->.joinDate	DateTime	

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

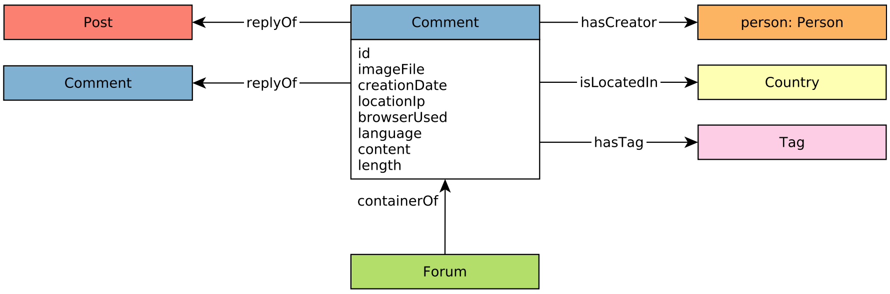
Interactive / update / 6

query	Interactive / update / 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	see below

1	`Post.id`	ID	
2	`Post.imageFile`	String	
3	`Post.creationDate`	DateTime	
4	`Post.locationIp`	String	
5	`Post.browserUsed`	String	
6	`Post.language`	String	
7	`Post.content`	Text	
8	`Post.length`	32-bit Integer	
9	`Post-hasCreator->Person.id`	ID	
10	`Post<-containerOf-Forum.id`	ID	
11	`Post-isLocatedIn->Country.id`	ID	
12	`Post-hasTag->Tag.id`	{ID}	

Interactive / update / 7

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

query	Interactive / update / 7
title	Add Comment
pattern	
desc.	Add a Comment **node** replying to a Post/Comment, connected to the network by 5 possible **edge** types (replyOf, hasCreator, containerOf, isLocatedIn, hasTag).

params

1	`Comment.id`	ID	
2	`Comment.creationDate`	DateTime	
3	`Comment.locationIp`	String	
4	`Comment.browserUsed`	String	
5	`Comment.content`	Text	
6	`Comment.length`	32-bit Integer	
7	`Comment-hasCreator->Person.id`	ID	
8	`Comment-isLocatedIn->Country.id`	ID	
9	`Comment-replyOf->Post.id`	ID	-1 if the comment is a reply of a comment
10	`Comment-replyOf->Comment.id`	ID	-1 if the comment is a reply of a post
11	`Comment-hasTag->Tag.id`	{ID}	

Interactive / update / 8

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

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

params

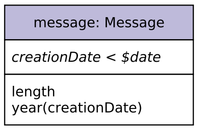
1	`Person.id`	ID	Person 1
2	`Person.id`	ID	Person 2
3	`Person-knows->.creationDate`	DateTime	

5 Business Intelligence Workload

5.1 Read Query Descriptions

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
BI 21
BI 22
BI 23
BI 24
BI 25

query	BI / read / 1
title	Posting summary
pattern	
desc.	Given a date, find all Messages created before that date. 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 | `date` | Date

result:

1	`year`	32-bit Integer	R	`year(message.creationDate)`
2	`isComment`	Boolean	M	`true` for Comments, `false` for Posts
3	`lengthCategory`	String	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 Integer	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. the ordering puts Posts first, and Comments second
3	`lengthCategory`	↑	order based on the length of the category, `0` (short), `1` (one liner), etc.

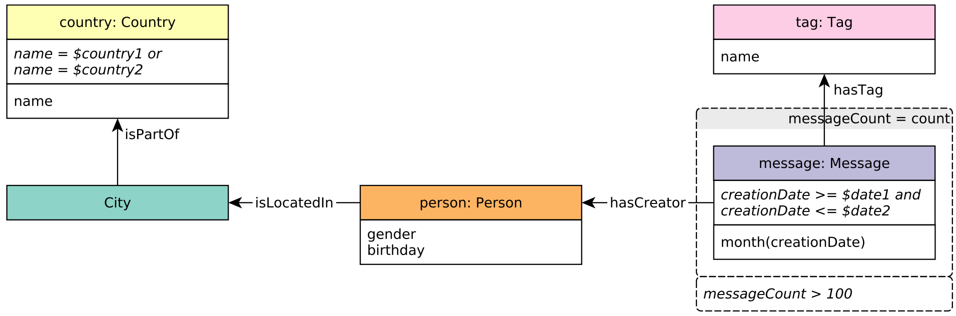
CPs	1.2, 3.2, 4.1, 8.5

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
BI 21
BI 22
BI 23
BI 24
BI 25

BI / read / 2

query	BI / read / 2
title	Top tags for country, age, gender, time

pattern



desc.

Select all Messages created in the range of `[startDate, endDate]` by Persons located in `country1` or `country2`. Select the creator Persons and the Tags of these Messages. Split these Persons, Tags and Messages into a 5-level grouping:

1. name of country of Person,
2. month the Message was created,
3. gender of Person,
4. age group of Person, defined as years between person's birthday and end of simulation (2013-01-01), divided by 5, rounded down (partial years do not count),
5. name of tag attached to Message.

Consider only those groups where number of Messages is greater than 100.

params

1	`startDate`	Date	
2	`endDate`	Date	
3	`country1`	String	
4	`country2`	String	

result

1	`country.name`	String	R	
2	`messageMonth`	32-bit Integer	R	1–12
3	`person.gender`	String	R	`male`/`female`
4	`ageGroup`	32-bit Integer	C	
5	`tag.name`	String	R	
6	`messageCount`	64-bit Integer	A	The number of messages in the group

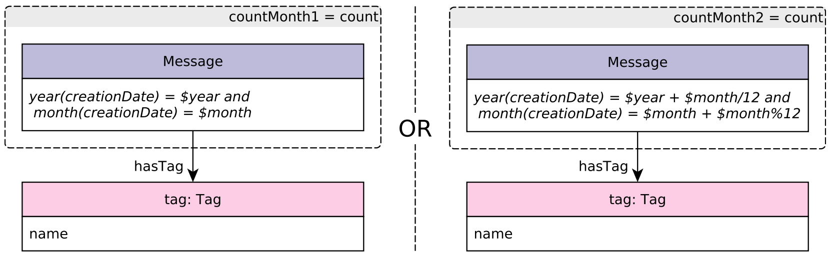
sort

1	`messageCount`	↓	
2	`tag.name`	↑	
3	`ageGroup`	↑	
4	`person.gender`	↑	
5	`messageMonth`	↑	
6	`country.name`	↑	

limit	100
CPs	1.1, 1.2, 1.3, 2.1, 2.3, 3.1, 3.2, 8.2, 8.5

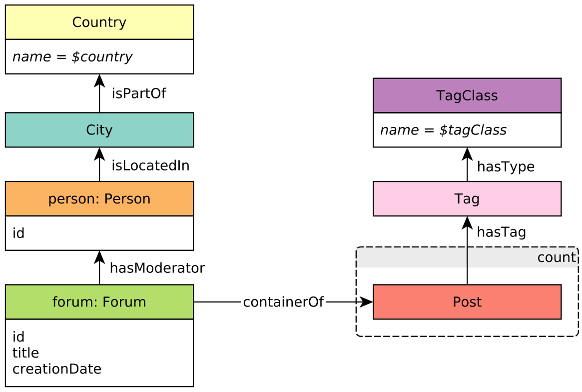
BI 1
BI 2
BI 3
BI 4
BI 5
BI 6
BI 7
BI 8
BI 9
BI 10
BI 11
BI 12
BI 13
BI 14
BI 15
BI 16
BI 17
BI 18
BI 19
BI 20
BI 21
BI 22
BI 23
BI 24
BI 25

BI / read / 3

<table>
<tr><td>query</td><td>BI / read / 3</td></tr>
<tr><td>title</td><td>Tag evolution</td></tr>
<tr><td>pattern</td><td>
</td></tr>
<tr><td>desc.</td><td>Find the Tags that were used in Messages during the given <code>month</code> of the given <code>year</code> and the Tags that were used during the next month.
For the Tags and for both months, compute the count of Messages.</td></tr>
<tr><td>params</td><td><table>
<tr><td>1</td><td><code>year</code></td><td>32-bit Integer</td><td></td></tr>
<tr><td>2</td><td><code>month</code></td><td>32-bit Integer</td><td></td></tr>
</table></td></tr>
<tr><td>result</td><td><table>
<tr><td>1</td><td><code>tag.name</code></td><td>String</td><td>R</td><td></td></tr>
<tr><td>2</td><td><code>countMonth1</code></td><td>32-bit Integer</td><td>A</td><td>Occurrences of the tag during the given <code>year</code> and <code>month</code></td></tr>
<tr><td>3</td><td><code>countMonth2</code></td><td>32-bit Integer</td><td>A</td><td>Occurrences of the tag during the next month after the given <code>year</code> and <code>month</code></td></tr>
<tr><td>4</td><td><code>diff</code></td><td>32-bit Integer</td><td>A</td><td>Absolute difference of <code>countMonth1</code> and <code>countMonth2</code></td></tr>
</table></td></tr>
<tr><td>sort</td><td><table>
<tr><td>1</td><td><code>diff</code></td><td>↓</td><td></td></tr>
<tr><td>2</td><td><code>tag.name</code></td><td>↑</td><td></td></tr>
</table></td></tr>
<tr><td>limit</td><td>100</td></tr>
<tr><td>CPs</td><td>2.4, 3.1, 3.2, 4.1, 4.3, 5.3, 6.1, 8.2, 8.5</td></tr>
</table>

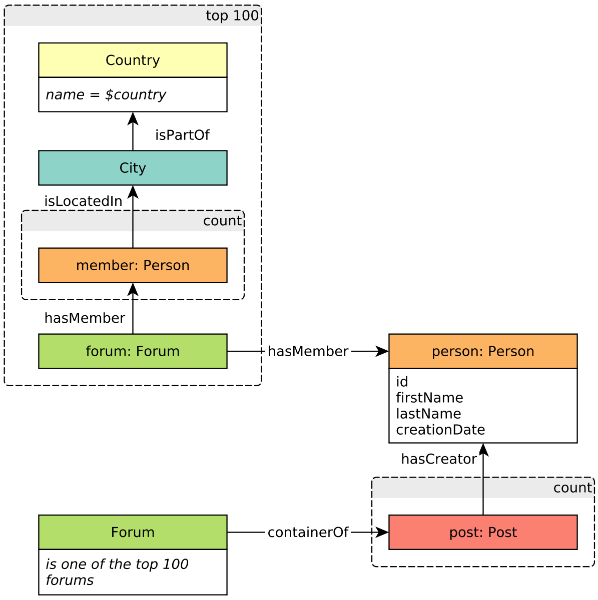
BI 1
BI 2
BI 3
BI 4
BI 5
BI 6
BI 7
BI 8
BI 9
BI 10
BI 11
BI 12
BI 13
BI 14
BI 15
BI 16
BI 17
BI 18
BI 19
BI 20
BI 21
BI 22
BI 23
BI 24
BI 25

BI / read / 4

query	BI / read / 4
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 Post with Tags belonging directly to the given TagClass. The location of a Forum is identified by the location of the Forum's moderator.
params	1 | `tagClass` | String 2 | `country` | String
result	1 | `forum.id` | 64-bit Integer | R 2 | `forum.title` | String | R 3 | `forum.creationDate` | DateTime | R 4 | `person.id` | 64-bit Integer | R 5 | `postCount` | 32-bit Integer | A
sort	1 | `postCount` | ↓ 2 | `forum.id` | ↑
limit	20
CPs	1.1, 1.2, 1.3, 2.1, 2.2, 2.4, 3.3, 8.2

BI / read / 5

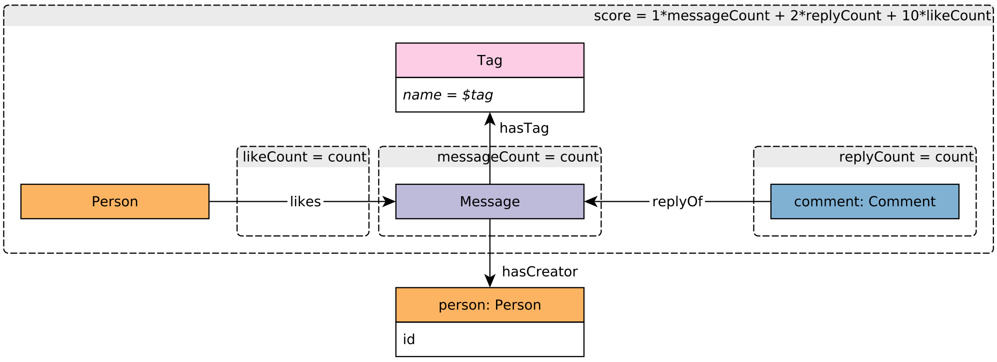
BI 1
BI 2
BI 3
BI 4
BI 5
BI 6
BI 7
BI 8
BI 9
BI 10
BI 11
BI 12
BI 13
BI 14
BI 15
BI 16
BI 17
BI 18
BI 19
BI 20
BI 21
BI 22
BI 23
BI 24
BI 25

query	BI / read / 5
title	Top posters in a country
pattern	
desc.	Find the most popular Forums for a given Country, where the popularity of a Forum is measured by the number of members that Forum has from the given Country. Calculate the top 100 most popular Forums. 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 Posts (`postCount`) they made in any of those (most popular) Forums. Also include those member Persons who have not posted any messages (have a `postCount` of 0).
params	1 `country` String
result	1 `person.id` 64-bit Integer R 2 `person.firstName` String R 3 `person.lastName` String R 4 `person.creationDate` DateTime R 5 `postCount` 32-bit Integer A
sort	1 `postCount` ↓ 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 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
BI 21
BI 22
BI 23
BI 24
BI 25

BI / read / 6

<table>
<tr><td>query</td><td>BI / read / 6</td></tr>
<tr><td>title</td><td>Most active Posters of a given Topic</td></tr>
<tr><td>pattern</td><td>



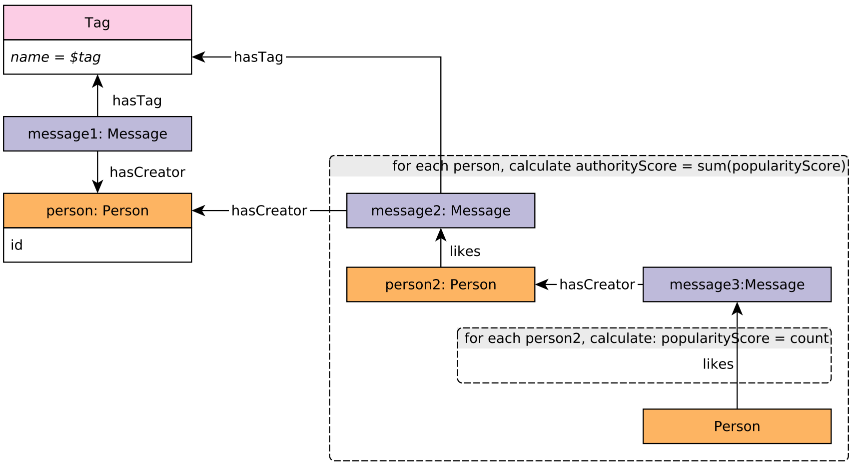
</td></tr>
<tr><td>desc.</td><td>Get each Person (person) who has created a Message (message) with a given Tag (direct relation, not transitive). Considering only these messages, for each Person node:

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 * messageCount + 2 * replyCount + 10 * likeCount.</td></tr>
<tr><td>params</td><td>
<table><tr><td>1</td><td>tag</td><td>String</td><td></td></tr></table>
</td></tr>
<tr><td>result</td><td>
<table>
<tr><td>1</td><td>person.id</td><td>64-bit Integer</td><td>R</td><td></td></tr>
<tr><td>2</td><td>replyCount</td><td>32-bit Integer</td><td>A</td><td></td></tr>
<tr><td>3</td><td>likeCount</td><td>32-bit Integer</td><td>A</td><td></td></tr>
<tr><td>4</td><td>messageCount</td><td>32-bit Integer</td><td>A</td><td></td></tr>
<tr><td>5</td><td>score</td><td>32-bit Integer</td><td>A</td><td></td></tr>
</table>
</td></tr>
<tr><td>sort</td><td>
<table>
<tr><td>1</td><td>score</td><td>↓</td><td></td></tr>
<tr><td>2</td><td>person.id</td><td>↑</td><td></td></tr>
</table>
</td></tr>
<tr><td>limit</td><td>100</td></tr>
<tr><td>CPs</td><td>1.2, 2.3, 8.2</td></tr>
</table>

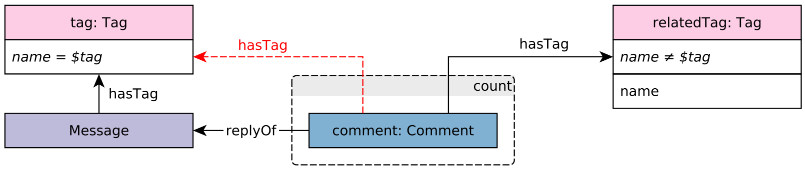
BI / read / 7

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
BI 21
BI 22
BI 23
BI 24
BI 25

query	BI / read / 7
title	Most authoritative users on a given topic
pattern	
desc.	Given a Tag, find all Persons (`person`) that ever created a Message (`message1`) with the given 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 (`person2`) that liked any of that Person's Messages (`message2`) with the given Tag. • A Person's (`person2`) "popularity score" is defined as the total number of likes on all of their Messages (`message3`).
params	1 | `tag` | String
result	1 | `person.id` | 64-bit Integer | R 2 | `authorityScore` | 32-bit Integer | A
sort	1 | `authorityScore` | ↓ 2 | `person1.id` | ↑
limit	100
CPs	1.2, 2.3, 3.2, 3.3, 6.1, 8.2

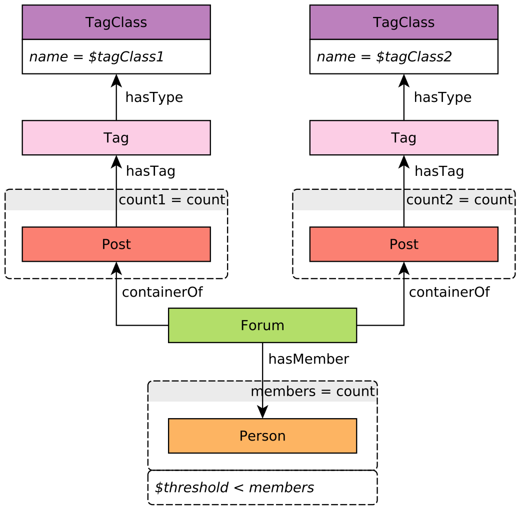
BI 1 BI 2 BI 3 BI 4 BI 5 BI 6 BI 7 BI 8 BI 9 BI 10 BI 11 BI 12 BI 13 BI 14 BI 15 BI 16 BI 17 BI 18 BI 19 BI 20 BI 21 BI 22 BI 23 BI 24 BI 25

BI / read / 8

query	BI / read / 8
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 | String
result	1 | relatedTag.name | 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 / 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
BI 21
BI 22
BI 23
BI 24
BI 25

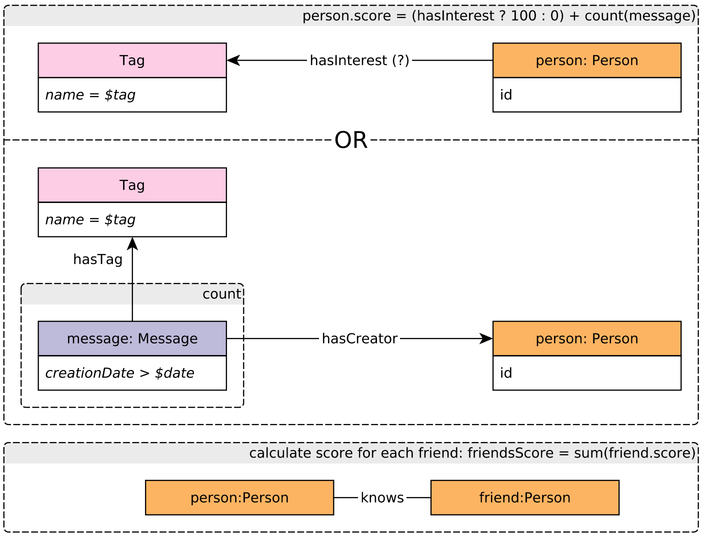
<table>
<tr><td>query</td><td>BI / read / 9</td></tr>
<tr><td>title</td><td>Forum with related Tags</td></tr>
<tr><td>pattern</td><td>
</td></tr>
<tr><td>desc.</td><td>Given two TagClasses (<code>tagClass1</code> and <code>tagClass2</code>), find Forums that contain

at least one Post (<code>post1</code>) with a Tag with a (direct) type of <code>tagClass1</code> and
at least one Post (<code>post2</code>) with a Tag with a (direct) type of <code>tagClass2</code>.

The <code>post1</code> and <code>post2</code> nodes may be the same Post.
Consider the Forums with a number of members greater than a given <code>threshold</code>. For every such Forum, count the number of <code>post1</code> nodes (<code>count1</code>) and the number of <code>post2</code> nodes (<code>count2</code>).</td></tr>
<tr><td>params</td><td>
<table>
<tr><td>1</td><td><code>tagClass1</code></td><td>String</td><td></td></tr>
<tr><td>2</td><td><code>tagClass2</code></td><td>String</td><td></td></tr>
<tr><td>3</td><td><code>threshold</code></td><td>32-bit Integer</td><td></td></tr>
</table></td></tr>
<tr><td>result</td><td>
<table>
<tr><td>1</td><td><code>forum.id</code></td><td>64-bit Integer</td><td>R</td><td></td></tr>
<tr><td>2</td><td><code>count1</code></td><td>32-bit Integer</td><td>A</td><td>Number of <code>post1</code> nodes</td></tr>
<tr><td>3</td><td><code>count2</code></td><td>32-bit Integer</td><td>A</td><td>Number of <code>post2</code> nodes</td></tr>
</table></td></tr>
<tr><td>sort</td><td>
<table>
<tr><td>1</td><td><code>abs(count2 - count1)</code></td><td>↓</td><td></td></tr>
<tr><td>2</td><td><code>forum.id</code></td><td>↑</td><td></td></tr>
</table></td></tr>
<tr><td>limit</td><td>100</td></tr>
<tr><td>CPs</td><td>1.2, 1.3, 2.1, 2.3, 2.4, 8.2</td></tr>
</table>

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 BI 21 BI 22 BI 23 BI 24 BI 25

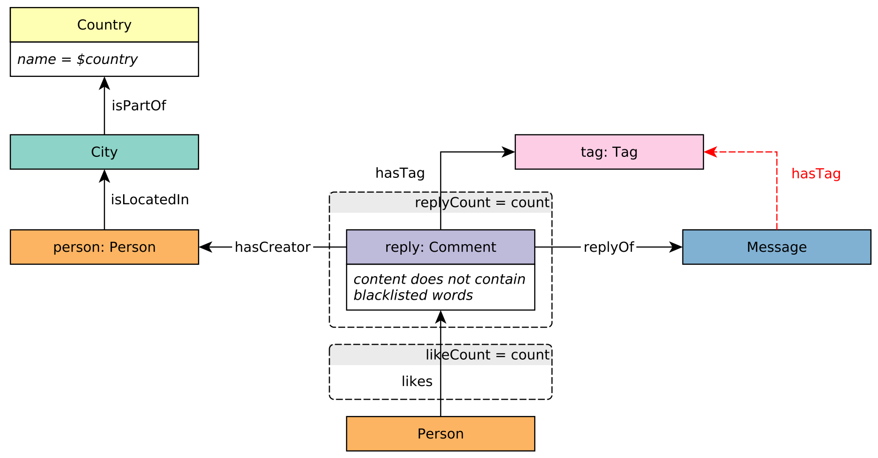
query	BI / read / 10
title	Central Person for a Tag
pattern	
desc.	Given a Tag, find all Persons that are interested in the Tag and/or have written a Message (Post or Comment) with a `creationDate` after a given `date` and that has a given Tag. For each Person, compute the `score` as the sum of the following 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` String 2 `date` Date
result	1 `person.id` 64-bit Integer 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, 8.2, 8.4, 8.5

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
BI 21
BI 22
BI 23
BI 24
BI 25

query	BI / read / 11
title	Unrelated replies

pattern



desc.

Find those Persons of a given Country that replied to any Message, such that the reply does not have any Tag in common with the Message (only direct replies are considered, transitive ones are not). Consider only those replies that do no contain any word from a given `blacklist`. For each Person and valid reply, retrieve the Tags associated with the reply, and retrieve the number of likes on the reply.

The detailed conditions for checking blacklisted words are currently as follows. Words do not have to stand separately, i.e. if the word "Green" is blacklisted, "South-Greenland" cannot be included in the results. Also, comparison should be done in a case-sensitive way. These conditions are preliminary and might be changed in later versions of the benchmark.

params

1	`country`	String	
2	`blacklist`	String[]	

result

1	`person.id`	64-bit Integer	R	
2	`tag.name`	String	R	
3	`likeCount`	32-bit Integer	A	The count of likes to replies with that Tag
4	`replyCount`	32-bit Integer	A	The count of replies with that Tag

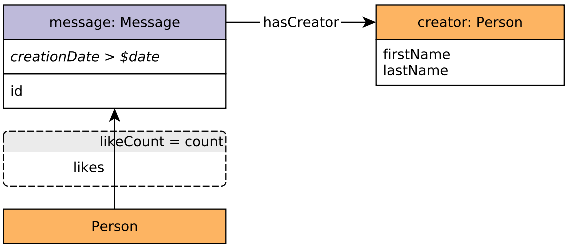
sort

1	`likeCount`	↓	
2	`person.id`	↑	
3	`tag.name`	↑	

limit	100
CPs	1.1, 2.1, 2.2, 2.3, 3.1, 3.2, 6.1, 8.1, 8.3

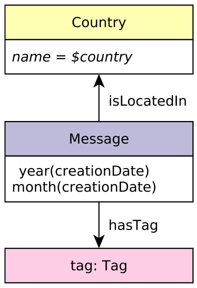
BI / read / 12

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 BI 21 BI 22 BI 23 BI 24 BI 25

<table>
<tr><td>query</td><td>BI / read / 12</td></tr>
<tr><td>title</td><td>Trending Posts</td></tr>
<tr><td>pattern</td><td>
</td></tr>
<tr><td>desc.</td><td>Find all Messages created after a given `date` (exclusive), that received more than a given number of likes (`likeThreshold`).</td></tr>
<tr><td>params</td><td><table>
<tr><td>1</td><td>date</td><td>Date</td><td></td></tr>
<tr><td>2</td><td>likeThreshold</td><td>32-bit Integer</td><td></td></tr>
</table></td></tr>
<tr><td>result</td><td><table>
<tr><td>1</td><td>message.id</td><td>64-bit Integer</td><td>R</td><td></td></tr>
<tr><td>2</td><td>message.creationDate</td><td>DateTime</td><td>R</td><td></td></tr>
<tr><td>3</td><td>creator.firstName</td><td>String</td><td>R</td><td>The first name of the Post's creator</td></tr>
<tr><td>4</td><td>creator.lastName</td><td>String</td><td>R</td><td>The last name of the Post's creator</td></tr>
<tr><td>5</td><td>likeCount</td><td>32-bit Integer</td><td>A</td><td>The number of likes the Post received</td></tr>
</table></td></tr>
<tr><td>sort</td><td><table>
<tr><td>1</td><td>likeCount</td><td>↓</td><td></td></tr>
<tr><td>2</td><td>message.id</td><td>↑</td><td></td></tr>
</table></td></tr>
<tr><td>limit</td><td>100</td></tr>
<tr><td>CPs</td><td>1.2, 2.2, 3.1, 6.1, 8.5</td></tr>
</table>

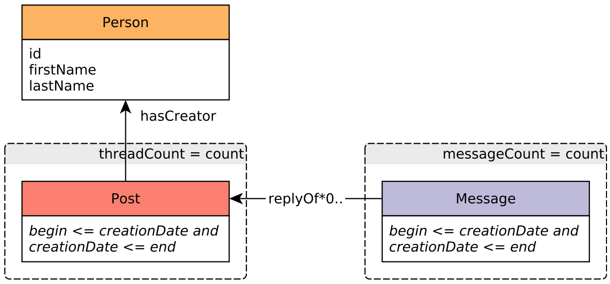
BI / read / 13

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 BI 21 BI 22 BI 23 BI 24 BI 25

query	BI / read / 13
title	Popular Tags per month in a country
pattern	
desc.	Find all Messages in a given Country, as well as their Tags. Group Messages by creation `year` and `month`. For each group, find the 5 most popular Tags, where popularity is the number of Messages (from within the same group) where the Tag appears. Note: even if there are no Tags for Messages in a given `year` and `month`, the result should include the `year` and `month` with an empty `popularTags` list.
params	1 | `country` | String
result	1 | `year` | 32-bit Integer | C | `year(message.creationDate)` 2 | `month` | 32-bit Integer | C | `month(message.creationDate)` 3 | `popularTags` | TagPairs | C | `(tag.name [String], popularity [32-bit Integer])` pairs, sorted descending by `popularity`, then ascending by `tag.name`
sort	1 | `year` | ↓ 2 | `month` | ↑
limit	100
CPs	1.2, 2.2, 2.3, 3.2, 6.1, 8.3, 8.5

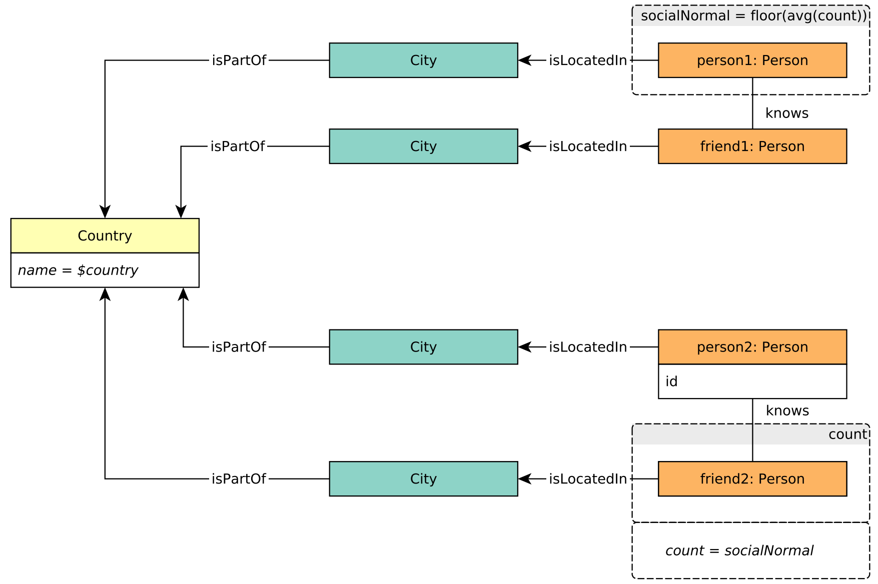
BI / read / 14

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 BI 21 BI 22 BI 23 BI 24 BI 25

<table>
<tr><td>query</td><td>BI / read / 14</td></tr>
<tr><td>title</td><td>Top thread initiators</td></tr>
<tr><td>pattern</td><td>
</td></tr>
<tr><td>desc.</td><td>For each Person, count the number of Posts they created in the time interval <code>[startDate, endDate]</code> (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.</td></tr>
<tr><td>params</td><td><table>
<tr><td>1</td><td><code>startDate</code></td><td>Date</td><td></td></tr>
<tr><td>2</td><td><code>endDate</code></td><td>Date</td><td></td></tr>
</table></td></tr>
<tr><td>result</td><td><table>
<tr><td>1</td><td><code>person.id</code></td><td>64-bit Integer</td><td>R</td><td></td></tr>
<tr><td>2</td><td><code>person.firstName</code></td><td>String</td><td>R</td><td></td></tr>
<tr><td>3</td><td><code>person.lastName</code></td><td>String</td><td>R</td><td></td></tr>
<tr><td>4</td><td><code>threadCount</code></td><td>32-bit Integer</td><td>A</td><td>The number of Posts created by that Person (the number of threads initiated)</td></tr>
<tr><td>5</td><td><code>messageCount</code></td><td>32-bit Integer</td><td>A</td><td>The number of Messages created in all the threads this Person initiated</td></tr>
</table></td></tr>
<tr><td>sort</td><td><table>
<tr><td>1</td><td><code>messageCount</code></td><td>↓</td><td></td></tr>
<tr><td>2</td><td><code>person.id</code></td><td>↑</td><td></td></tr>
</table></td></tr>
<tr><td>limit</td><td>100</td></tr>
<tr><td>CPs</td><td>1.2, 2.2, 2.3, 3.2, 7.2, 7.3, 7.4, 8.1, 8.5</td></tr>
</table>

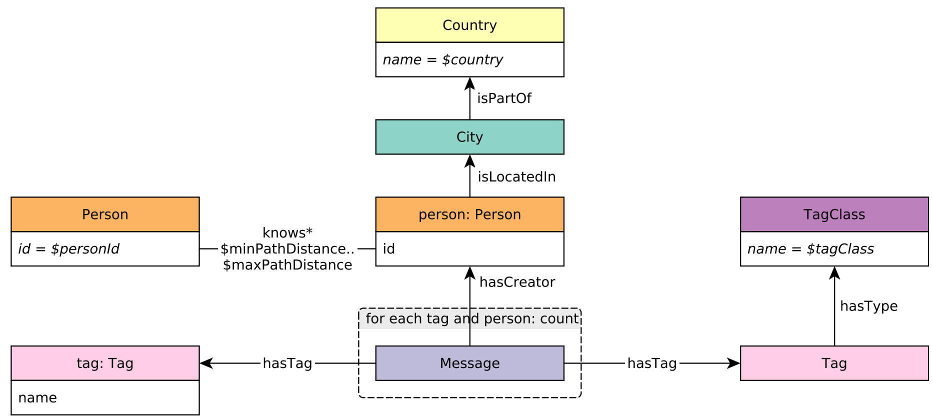
BI / read / 15

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 BI 21 BI 22 BI 23 BI 24 BI 25

query	BI / read / 15
title	Social normals
pattern	
desc.	Given a Country `country`, determine the "social normal", i.e. the floor of average number of friends that Persons of `country` have in `country`. Then, find all Persons in `country`, whose number of friends in `country` equals the social normal value.
params	1 `country` String
result	1 `person.id` 64-bit Integer R 2 `count` 32-bit Integer A
sort	1 `person.id` ↑
limit	100
CPs	1.2, 2.3, 3.2, 3.3, 5.3, 6.1, 8.2, 8.4

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
BI 21
BI 22
BI 23
BI 24
BI 25

BI / read / 16

query	BI / read / 16
title	Experts in social circle
pattern	

desc.

Given a Person, find all other Persons that live in a given `country` and are connected to given Person by a transitive path with length in range `[minPathDistance, maxPathDistance]` through the knows relation.
In the path, an edge can be only traversed once while nodes can be traversed multiple times.
For each of these Persons, 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`	64-bit Integer	
2	`country`	String	
3	`tagClass`	String	
4	`minPathDistance`	32-bit Integer	
5	`maxPathDistance`	32-bit Integer	

result

1	`person.id`	64-bit Integer	R	
2	`tag.name`	String	R	
3	`messageCount`	32-bit Integer	A	Number of Messages created by that Person containing that Tag

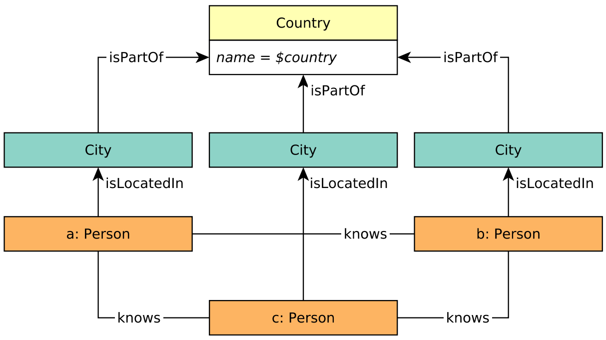
sort

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

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

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
BI 21
BI 22
BI 23
BI 24
BI 25

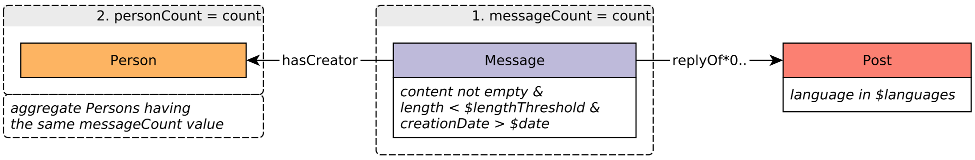
query	BI / read / 17
title	Friend triangles
pattern	
desc.	For a given `country`, count all the distinct triples of Persons such that: • `a` is friend of `b`, • `b` is friend of `c`, • `c` is friend of `a`. Distinct means that given a triple t_1 in the result set R of all qualified triples, there is no triple t_2 in R such that t_1 and t_2 have the same set of elements.
params	1 | `country` | String
result	1 | `count` | 32-bit Integer | A
CPs	1.1, 2.3

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
BI 21
BI 22
BI 23
BI 24
BI 25

BI / read / 18

query	BI / read / 18
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, `imageFile` 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	
3	`languages`	String[]	

result

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

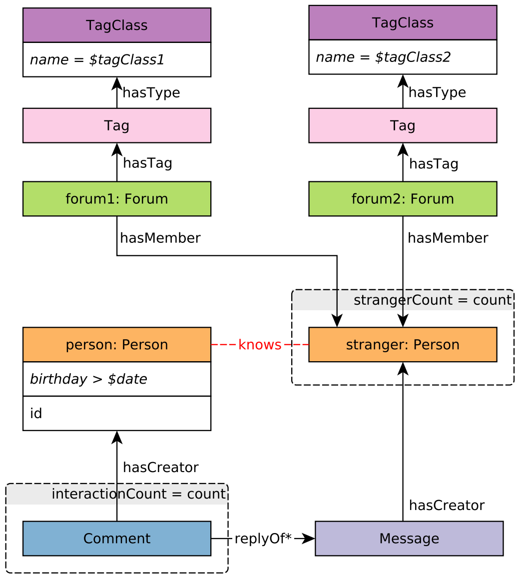
sort

1	`personCount`	↓	
2	`messageCount`	↓	

CPs	1.1, 1.2, 1.4, 3.2, 4.2, 4.3, 8.1, 8.2, 8.3, 8.4, 8.5

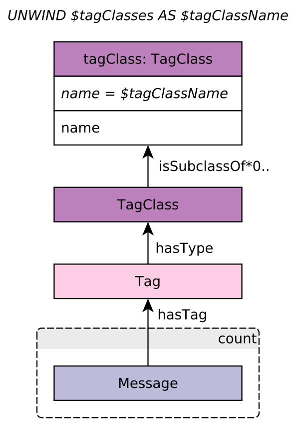
BI / read / 19

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 BI 21 BI 22 BI 23 BI 24 BI 25

<table>
<tr><td>query</td><td>BI / read / 19</td></tr>
<tr><td>title</td><td>Stranger’s interaction</td></tr>
<tr><td>pattern</td><td>
</td></tr>
<tr><td>desc.</td><td>For all the Persons (person) born after a certain date, find all the strangers they interacted with, where strangers are Persons that do not know person. There is no restriction on the date that strangers were born.
Consider only strangers that are
members of Forums tagged with a Tag with a (direct) type of tagClass1 and
members of Forums tagged with a Tag with a (direct) type of tagClass2.
The Tags may be attached to the same Forum or they may be attached to different Forums.
Interaction is defined as follows: the person has replied to a Message by the stranger B (the reply might be a transitive one).
For each person, count the number of strangers they interacted with (strangerCount) and total number of times they interacted with them (interactionCount).</td></tr>
<tr><td>params</td><td><table><tr><td>1</td><td>date</td><td>Date</td><td></td></tr><tr><td>2</td><td>tagClass1</td><td>String</td><td></td></tr><tr><td>3</td><td>tagClass2</td><td>String</td><td></td></tr></table></td></tr>
<tr><td>result</td><td><table><tr><td>1</td><td>person.id</td><td>64-bit Integer</td><td>R</td><td></td></tr><tr><td>2</td><td>strangerCount</td><td>32-bit Integer</td><td>A</td><td></td></tr><tr><td>3</td><td>interactionCount</td><td>32-bit Integer</td><td>A</td><td></td></tr></table></td></tr>
<tr><td>sort</td><td><table><tr><td>1</td><td>interactionCount</td><td>↓</td><td></td></tr><tr><td>2</td><td>person.id</td><td>↑</td><td></td></tr></table></td></tr>
<tr><td>limit</td><td>100</td></tr>
<tr><td>CPs</td><td>1.1, 1.3, 2.1, 2.3, 2.4, 3.3, 5.1, 7.3, 7.4, 8.1, 8.5</td></tr>
</table>

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
BI 21
BI 22
BI 23
BI 24
BI 25

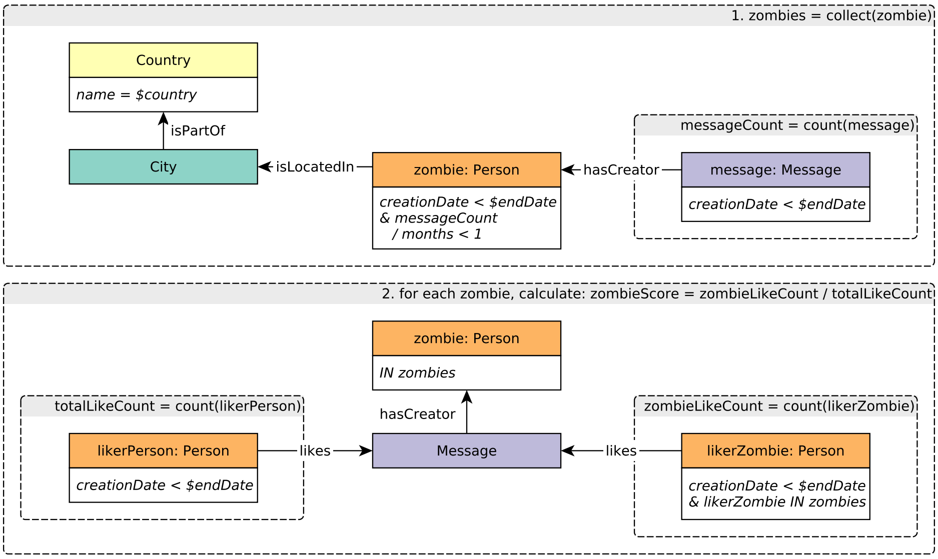
BI / read / 20

query	BI / read / 20				
title	High-level topics				
pattern					
desc.	For all given TagClasses, count number of Messages that have a Tag that belongs to that TagClass or any of its children (all descendants through a transitive relation).				
params	1	`tagClasses`	String[]		
result	1	`tagClass.name`	String	R	The TagClass of the root
	2	`messageCount`	32-bit Integer	A	
sort	1	`messageCount`	↓		
	2	`tagClass.name`	↑		
limit	100				
CPs	1.4, 2.1, 6.1, 8.1				

BI / read / 21

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 BI 21 BI 22 BI 23 BI 24 BI 25

<table>
<tr><td>query</td><td>BI / read / 21</td></tr>
<tr><td>title</td><td>Zombies in a country</td></tr>
<tr><td>pattern</td><td>



</td></tr>
<tr><td>desc.</td><td>

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.

For both `zombieLikeCount` and `totalLikeCount`, only consider likes received from profiles that were created before the given `endDate`.

</td></tr>
<tr><td>params</td><td>

1	country	String
2	endDate	Date

</td></tr>
<tr><td>result</td><td>

1	zombie.id	64-bit Integer	R	
2	zombieLikeCount	32-bit Integer	A	
3	totalLikeCount	32-bit Integer	A	
4	zombieScore	64-bit Float	A	zombieLikeCount / totalLikeCount

</td></tr>
<tr><td>sort</td><td>

1	zombieScore	↓
2	zombie.id	↑

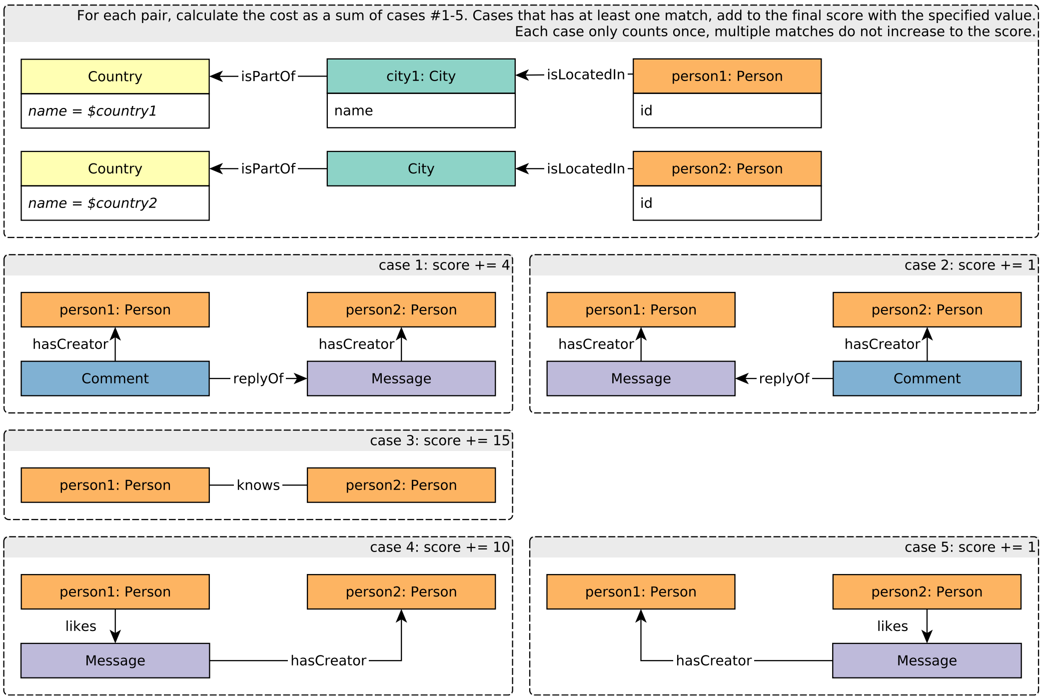
</td></tr>
<tr><td>limit</td><td>100</td></tr>
<tr><td>CPs</td><td>1.2, 2.1, 2.3, 2.4, 3.2, 3.3, 5.1, 5.3, 8.2, 8.4, 8.5</td></tr>
</table>

BI / read / 22

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 BI 21 BI 22 BI 23 BI 24 BI 25

query	BI / read / 22
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 Comment 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`.
To break ties, order by (1) `person1.id` ascending and (2) `person2.id` ascending.

params

1	country1	String
2	country2	String

result

1	person1.id	64-bit Integer	R
2	person2.id	64-bit Integer	R
3	city1.name	String	R
4	score	32-bit Integer	C

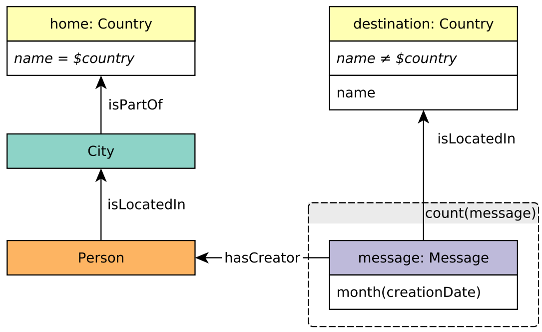
sort

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

CPs 1.3, 1.4, 2.1, 3.1, 3.3, 5.1, 5.2, 5.3, 8.3, 8.4

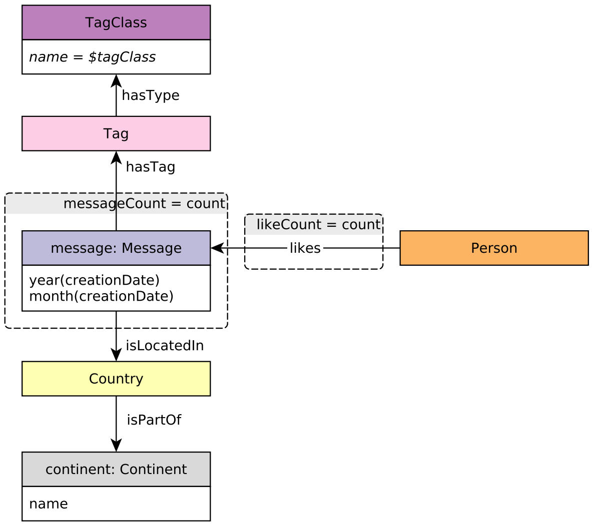
BI / read / 23

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 BI 21 BI 22 BI 23 BI 24 BI 25

query	BI / read / 23
title	Holiday destinations
pattern	
desc.	Count the Messages of all residents of a given Country (`home`), where the message was written abroad. Group the messages by `month` and `destination`. A Message was written abroad if it is located in a Country (`destination`) different than `home`.
params	1 · `country` · String
result	1 · `messageCount` · 32-bit Integer · A · The number of Messages in each group 2 · `destination.name` · String · R · The name of the destination Country 3 · `month` · 32-bit Integer · C · month(message.creationDate)
sort	1 · `messageCount` · ↓ 2 · `desination.name` · ↑ 3 · `month` · ↑
limit	100
CPs	1.4, 2.3, 2.4, 3.3, 4.3, 8.5

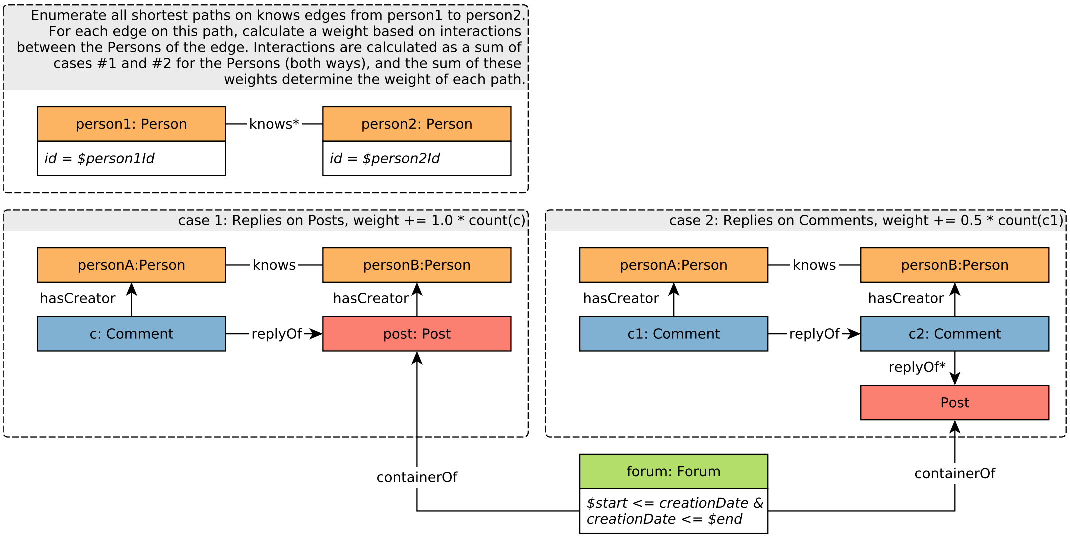
BI 1 BI 2 BI 3 BI 4 BI 5 BI 6 BI 7 BI 8 BI 9 BI 10 BI 11 BI 12 BI 13 BI 14 BI 15 BI 16 BI 17 BI 18 BI 19 BI 20 BI 21 BI 22 BI 23 BI 24 BI 25

BI / read / 24

query	BI / read / 24
title	Messages by Topic and Continent
pattern	
desc.	Find all Messages tagged with a Tag that has the (direct) type of the given `tagClass`. Count all Messages and their likes grouped by Continent, `year`, and `month`.
params	1 · `tagClass` · String
result	1 · `messageCount` · 32-bit Integer · A 2 · `likeCount` · 32-bit Integer · A 3 · `year` · 32-bit Integer · C · `year(message.creationDate)` 4 · `month` · 32-bit Integer · C · `month(message.creationDate)` 5 · `continent.name` · String · R
sort	1 · `year` · ↑ 2 · `month` · ↑ 3 · `continent.name` · ↓
limit	100
CPs	1.4, 2.1, 2.3, 2.4, 3.2, 4.3, 8.5

BI / read / 25

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
BI 21
BI 22
BI 23
BI 24
BI 25

<table>
<tr><td>query</td><td>BI / read / 25</td></tr>
<tr><td>title</td><td>Weighted interaction paths</td></tr>
<tr><td>pattern</td><td>
</td></tr>
<tr><td>desc.</td><td>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 <code>[startDate, endDate]</code>. Note that for Comments, the containing Forum is that of the Post that the comment (transitively) replies to.
Return all paths with the Person ids ordered by their weights descending.</td></tr>
<tr><td>params</td><td><table>
<tr><td>1</td><td><code>person1Id</code></td><td>64-bit Integer</td><td></td></tr>
<tr><td>2</td><td><code>person2Id</code></td><td>64-bit Integer</td><td></td></tr>
<tr><td>3</td><td><code>startDate</code></td><td>Date</td><td></td></tr>
<tr><td>4</td><td><code>endDate</code></td><td>Date</td><td></td></tr>
</table></td></tr>
<tr><td>result</td><td><table>
<tr><td>1</td><td><code>person.id</code></td><td>64-bit Integer[]</td><td>R</td><td>Identifiers representing an ordered sequence of the Persons in the path weight</td></tr>
<tr><td>2</td><td><code>weight</code></td><td>64-bit Double</td><td>R</td><td></td></tr>
</table></td></tr>
<tr><td>sort</td><td><table>
<tr><td>1</td><td><code>weight</code></td><td>↓</td><td></td></tr>
<tr><td>2</td><td><code>personIds</code></td><td>↑</td><td>The ids in the paths are used for lexicographical sorting</td></tr>
</table></td></tr>
<tr><td>CPs</td><td>1.2, 2.1, 2.2, 2.4, 3.3, 5.1, 5.3, 7.2, 7.3, 8.1, 8.2, 8.3, 8.4, 8.5, 8.6</td></tr>
</table>

5.2 Update Query Descriptions

The BI workload currently does not define update operations. However, this is subject to future research and the task force is working on determining possible update scenarios.

6 Auditing Rules

This chapter describes the rules to audit benchmark runs, that is, what techniques are allowed and what are not, what must be provided to the auditor and guidelines for the auditors to perform the audit.

6.1 Preparation

The first step when doing an audit is to determine the versions of the following items that will be used for the benchmark:

- The benchmark specification
- The data generator
- The driver

These 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 scale factor must be provided, including the update streams and substitution parameters.

6.1.1 Collect System Details

The next step is to collect the technical and pricing details of the system under test. This includes the following items:

- Common name of the system, e.g. Dell PowerEdge xxxx.
- Type and number of CPUs, cores/threads per CPU, clock frequency and cache hierarchy characteristics (levels, size per level, etc.).
- The amount of the system's memory, type and frequency.
- The disk controller or motherboard type if disk controller is on the motherboard.
- For each distinct type of secondary storage device, the number and characteristics of the device.
- The number and type of network controllers.
- The number and type of network switches. Wiring must be disclosed.
- Date of availability of the system.

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, no network need be reported. The price of the hardware in question must be disclosed and 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

Besides hardware characteristics, also software details must be collected:

- The DBMS and operating system name and versions.
- Installation and configuration information of both the DBMS and operating system, which must be provided by the test sponsor.
- Price of the software license used, which can be tied to the number of concurrent users or size of data.
- Date of availability of the software.

Also, the test sponsor must provide all the source code relevant to the benchmark.

6.1.2 Setup the Benchmark Environment

Once all the information has been collected, the auditor will setup the environment to perform the benchmark run. This setup includes configuring the following items:

- Setup the LDBC Data generator in the test machine if datasets are not available from a trusted source.
- Setup the LDBC driver with the connectors provided by the test sponsor. The test sponsor must provide the configuration parameters to configure the driver (tcr, number of threads, etc.).
  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. 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. These can found in Section B.1. If the driver will be executed on a separate machine, gather the characteristics of that machine in the same way as specified above.

6.1.3 Load Data

The test sponsor must provide all the necessary documentation and scripts to load the dataset into the database to test. 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. The auditor will measure the time to load the data, which will be disclosed.

6.2 Running the Benchmark

Running the benchmark consists of three separate parts: (1) validating the query implementations, (2) warming the database and (3) performing the benchmark run. The queries are validated by means of the official validation datasets provided by LDBC consortium in their official software repositories. 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 warmup can be performed either using the LDBC driver or externally, and the way it is performed must be disclosed.

A valid benchmark run must last at least 2 hours of simulation time (datagen time). Also, in order to be valid, a benchmark run needs to meet the following requirements. 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:

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

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

6.3 Recovery

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 warmup period, an execution of the benchmark will be performed under the same terms as in the previous measured run.

At an arbitrary point close to 2 hours of simulation execution time, the machine will be disconnected. Then, the auditor will restart the database system and will check that the last commited 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.

6.4 Serializability

Optionally, the test sponsor can execute update queries atomically. The auditor will verify that serializability is guaranteed.

7 Related Work

7.1 Graph Database Benchmarks

Train Benchmark: model validation [15].

Graph database comparison [11].

An “interactive social network benchmark”: [5].

XDGBench [6]

7.2 Scalable Graph Generators

The DATAGEN component is a fork of the generator described in [14].

7.3 LDBC Publications

LDBC publications are listed at `http://ldbcouncil.org/publications`.

References

[1] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5):68:1–68:40, Sept. 2017.

[2] R. Angles, P. A. Boncz, J. Larriba-Pey, I. Fundulaki, T. Neumann, O. Erling, P. Neubauer, N. Martínez-Bazan, V. Kotsev, and I. Toma. The Linked Data Benchmark Council: a graph and RDF industry benchmarking effort. *SIGMOD Record*, 43(1):27–31, 2014.

[3] R. Angles et al. G-CORE: A Core for Future Graph Query Languages. Dec. 2017.

[4] A. Averbuch and A. Prat-Pérez. Benchmark design for navigational pattern matching benchmarking. Technical report, Linked Data Benchmark Council, 2014. `http://ldbcouncil.org/sites/default/files/LDBC_D3.3.34.pdf`.

[5] S. Barahmand and S. Ghandeharizadeh. BG: A benchmark to evaluate interactive social networking actions. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, 2013.

[6] M. Dayarathna and T. Suzumura. Graph database benchmarking on cloud environments with XGDBench. *Autom. Softw. Eng.*, 21(4):509–533, 2014.

[7] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat-Pérez, M. Pham, and P. A. Boncz. The LDBC Social Network Benchmark: Interactive workload. In *SIGMOD*, pages 619–630, 2015.

[8] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[9] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundam. Inform.*, 26(3/4):287–313, 1996.

[10] A. Iosup, T. Hegeman, W. L. Ngai, S. Heldens, A. Prat-Pérez, T. Manhardt, H. Chafi, M. Capota, N. Sundaram, M. J. Anderson, I. G. Tanase, Y. Xia, L. Nai, and P. A. Boncz. LDBC Graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *PVLDB*, 9(13):1317–1328, 2016.

[11] M. Lissandrini, M. Brugnara, and Y. Velegrakis. An evaluation methodology and experimental comparison of graph databases. Technical report, University of Trento, 2017. `https://disi.unitn.it/~lissandrini/pdf/lissandrini-techreport-gdb.pdf`.

[12] G. Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *PVLDB*, pages 476–487, 1998.

[13] T. Neumann and G. Moerkotte. A framework for reasoning about share equivalence and its integration into a plan generator. In *BTW*, pages 7–26, 2009.

[14] M. Pham, P. A. Boncz, and O. Erling. S3G2: A scalable structure-correlated social graph generator. In *Selected Topics in Performance Evaluation and Benchmarking - 4th TPC Technology Conference, TPCTC 2012, Istanbul, Turkey, August 27, 2012, Revised Selected Papers*, pages 156–172, 2012.

[15] G. Szárnyas, B. Izsó, I. Ráth, and D. Varró. The Train Benchmark: Cross-technology performance evaluation of continuous model validation. *Softw Syst Model*, 2017.

A Choke Points

Introduction

Choke points are a superset of [4] with the exception of CP 7.1, which was removed and replaced with a new choke point.

The connection between choke points and queries is displayed in Table A.1.

	1.1	1.2	1.3	1.4	2.1	2.2	2.3	2.4	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	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				●	●													●					●					
BI 21		●			●		●	●		●	●				●		●							●		●	●	
BI 22			●	●	●				●		●				●	●	●								●	●		
BI 23				●			●	●			●			●													●	
BI 24				●	●		●	●		●				●													●	
BI 25		●			●	●		●			●				●		●			●	●		●	●	●	●	●	●
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 indexes 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 2 BI 4 BI 11 BI 17 BI 18 BI 19 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 2 BI 4 BI 5 BI 6 BI 7 BI 9 BI 10 BI 12 BI 13 BI 14 BI 15 BI 16 BI 18 BI 21 BI 25 IC 9

CP-1.3: [QOPT] Top-k push down

Top-k push down. 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. 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 2 BI 4 BI 5 BI 9 BI 16 BI 19 BI 22 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 8 BI 18 BI 20 BI 22 BI 23 BI 24

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 as 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, these chokepoint tests the ability of the query optimizer to find optimal join (traversal) orders.

Queries

`BI 2` `BI 4` `BI 5` `BI 9` `BI 10` `BI 11` `BI 19` `BI 20` `BI 21` `BI 22` `BI 24` `BI 25` `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 4` `BI 5` `BI 11` `BI 12` `BI 13` `BI 14` `BI 25` `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 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 2` `BI 5` `BI 6` `BI 7` `BI 9` `BI 10` `BI 11` `BI 13` `BI 14` `BI 15` `BI 16` `BI 17` `BI 19` `BI 21` `BI 23` `BI 24` `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 indexes or implementing Bloom filter tests inside the join are techniques to improve the performance in such situations [8].

Queries

`BI 3` `BI 4` `BI 5` `BI 9` `BI 16` `BI 19` `BI 21` `BI 23` `BI 24` `BI 25` `IC 8` `IC 11`

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 indexes, 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 indexes, 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 3` `BI 11` `BI 12` `BI 22` `IC 3`

CP-3.2: [STORAGE] Dimensional clustering

This chokepoint 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 3` `BI 7` `BI 10` `BI 11` `BI 13` `BI 14` `BI 15` `BI 18` `BI 21` `BI 24` `IC 2` `IC 8` `IC 9`

CP-3.3: [QEXE] Scattered index access patterns

This choke-point tests the performance of indexes 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 neighborhood traversal is an example of an operation with random access without predictable locality.

Queries

`BI 4` `BI 5` `BI 7` `BI 8` `BI 15` `BI 16` `BI 19` `BI 21` `BI 22` `BI 23` `BI 25` `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 average aggregates can be derived afterwards by dividing a SUM by the COUNT when those have been computed.

Queries

`BI 1` `BI 3` `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 [12].

Queries

`BI 18` `IC 10`

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 3` `BI 18` `BI 23` `BI 24`

CP-4.4: [QEXE] String matching performance

This chokepoint tests the ability of efficiently evaluating complex string matching expressions (e.g. via regular expressions).

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. To execute queries 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 block-wise 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 19` `BI 21` `BI 22` `BI 25` `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 [13] 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 8` `BI 22` `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 3` `BI 5` `BI 15` `BI 21` `BI 22` `BI 25` `IC 1` `IC 8`

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 3` `BI 5` `BI 7` `BI 11` `BI 12` `BI 13` `BI 15` `BI 20` `IC 10`

A.7 RDF and Graph Specifics

CP-7.1: [QEXE] Path pattern reuse

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 16` `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 neighborhood 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 14` `BI 16` `BI 25` `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 14` `BI 16` `BI 19` `BI 25` `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 14` `BI 19`

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 [9].

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
```

G-CORE. TODO

SPARQL. Prior to SPARQL 1.1, queries with negative edge conditions used the following syntax:

```
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 8 BI 11 BI 14 BI 16 BI 18 BI 19 BI 20 BI 25

[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: `https://www.w3.org/TR/sparql11-query/#neg-notexists-minus`

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]

G-CORE. TODO

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 7 BI 9 BI 10 BI 15 BI 18 BI 21 BI 25

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
...
```

G-CORE. TODO

SPARQL. TODO

Queries

BI 11 BI 13 BI 18 BI 22 BI 25

[3] `http://www.opencypher.org/blog/2017/07/27/ocig1-aggregations-blog/`

[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.

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.[5]
G-CORE. TODO
SPARQL. SPARQL fully supports query composition.[6]

Queries

`BI 5` `BI 10` `BI 15` `BI 18` `BI 21` `BI 22` `BI 25`

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 a month).

Language-specific notes

Cypher. The openCypher project has accepted a proposal to support dates and times.[7]
G-CORE. TODO
SPARQL. SPARQL supports dates and times extensively with timezones and functions for extracting a part of a given date.[8]

Queries

`BI 1` `BI 2` `BI 3` `BI 10` `BI 12` `BI 13` `BI 14` `BI 18` `BI 19` `BI 21` `BI 23` `BI 24` `BI 25`

CP-8.6: [LANG] Handling paths

Description

Note on terminology. The *Glossary of graph theory terms* page of Wikipedia[9] defines *paths* as follows: "A path may either be a walk (a sequence of vertices 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 vertices 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 [1].

Handling paths as first-class citizens is one of the key distinguishing features of graph database systems [3]. Hence, additionally to reachability-style checks, a language should be able to express queries that operate on

[5] https://github.com/petraselmer/openCypher/blob/1ca70bf8e3cea65ee47ce49eeabea83530eb529b/cip/1.accepted/CIP2016-06-22-nested-updating-and-chained-subqueries.adoc

[6] https://www.w3.org/TR/sparql11-query/#subqueries

[7] https://github.com/thobe/openCypher/blob/06accdb3e69820cdfac3dbd50a4f8eee73ba179a/cip/1.accepted/CIP2015-08-06-date-time.adoc

[8] https://www.w3.org/TR/sparql11-query/#func-date-time

[9] https://en.wikipedia.org/wiki/Glossary_of_graph_theory_terms

elements of a path, e.g. calculate a score on each edge of the path. Also, some use cases specify uniqueness constraints on paths, e.g. that a certain path must not have repeated nodes (referred to as "walks" in graph theory) or not have repeated edges ("trails" in graph theory). More precisely, paper [1] defines *homomorphism-based semantics* (no constraints on repetitions) and multiple flavours of *isomorphism-based semantics*:

- *no-repeated-node semantics* (also known as *fully isomorphic matching*),
- *no-repeated-edge semantics* (also known as *edge-isomorphic matching*),
- *no-repeated-anything semantics* (not used in the context of this work).

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.[10] RPQs are also proposed in the openCypher language as *path patterns*.[11]

G-CORE. The G-CORE language [3] 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*.

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 )`.

Queries

`BI 16` `BI 25`

[10] `https://github.com/boggle/openCypher/blob/c139130b49aebe6a85fc395e2cf03cfeec8484c6/cip/1.accepted/CIP2017-01-18-configurable-pattern-matching-semantics.adoc`

[11] `https://github.com/thobe/openCypher/blob/b95eec108ce4ec07eedfe13b3e5fff0e94f789a4/cip/1.accepted/CIP2017-02-06-Path-Patterns.adoc`

B Scale Factor Statistics

B.1 Scale Factor Statistics

Query Type	**SF1**	**SF3**	**SF10**	**SF30**	**SF100**	**SF300**	**SF1000**
Query 1	26	26	26	26	26	26	26
Query 2	37	37	37	37	37	37	37
Query 3	69	79	92	106	123	142	165
Query 4	36	36	36	36	36	36	36
Query 5	57	61	66	72	78	84	91
Query 6	129	172	236	316	434	580	796
Query 7	87	72	54	48	38	32	25
Query 8	45	27	15	9	5	3	1
Query 9	157	209	287	384	527	705	967
Query 10	30	32	35	37	40	44	47
Query 11	16	17	19	20	22	24	26
Query 12	44	44	44	44	44	44	44
Query 13	19	19	19	19	19	19	19
Query 14	49	49	49	49	49	49	49

Table B.1: Frequencies for each query and SF.