LDBC

Cooperative Project

FP7 – 317548

Benchmark design for navigational pattern matching benchmarking

Coordinator: [Arnau Prat (UPC), Alex Averbuch (NEO)]

With contributions from: [Peter Boncz (VUA), Marta Perez (UPC), Ricard Gavalda (UPC), Renzo Angles (TALCA), Orri Erling (OGL), Andrey Gubichev (TUM), Mirko Spasić (OGL), Minh-Duc Pham (VUA), Norbert Martínez (SPARSITY)]

1st Quality Reviewer: Irini Fundulaki (FORTH)
2nd Quality Reviewer: Andrey Gubichev (TUM)
3rd Quality Reviewer: Venelin Kotsev (OWLIM)

Deliverable nature:	Report (R)
Dissemination level: (Confidentiality)	Public (PU)
Contractual delivery date:	M24
Actual delivery date:	M24
Version:	1.0
Total number of pages:	102
Keywords:	benchmark, choke points, dataset generator, graph database, query set, RDF, workload, auditing rules, publication rules, scale factors

Abstract

This document discusses all the aspects of the design of a navigational benchmark, that is, a benchmark for testing small pattern matching queries that navigate around a small portion of the graph, starting from a given node. We enumerate the challenges these type of queries pose, what characteristics the data used by the benchmark needs to have, a set of chokepoints a query set proposal is architected around, and a set of performance metrics. It also includes query implementations for several vendor technologies.

Executive Summary

The demand for efficient graph data management technologies has emerged during the last years, due to the increasing number of graph applications taking advantage of the resources of the new data era. A large variety of use cases exist. One common denominator of many graph applications demand systems capable of answering short graph pattern matching queries in real time, what we define as interactive navigational queries. An interactive navigational query typically queries the graph starting from an input node, and explores the neighborhood of that node (2 to 3 hops in general).

In order to provide graph application developers with more objective ways to compare different vendor technologies for their needs, we require for representative benchmarks that mimic the actual workloads present in real applications.

In this document we describe the aspects that need to be taken into consideration when developing a benchmark for interactive navigational queries. This includes enumerating the challenges posed by these types of queries, proposing a set of choke points that drive the development of a workload consisting of challenging navigational queries, a description of the characteristics required by the data the benchmark is running over and a set of representative performance metrics. Finally, we include the set of queries implemented on several existing technologies, as well as some preliminary results.

From this design document, one of the workloads proposed by the LDBC-SNB benchmark has emerged, which is currently available online.

Document Information

IST Project Number	FP7 – 317548	Acronym	LDBC
Full Title	LDBC		
Project URL	http://www.ldbc.eu/		
Document URL	https://svn.sti2.at/ldbc/trunk/wp3/deliverables/D3.3.3_Benchmark_design_for_navigational_benchmarking		
EU Project Officer	Carola Carstens		

Deliverable	Number		Title	Benchmark design for navigational pattern matching benchmarking
Work Package	Number	N/A	Title	Social Network Benchmark Task Force

Date of Delivery	Contractual	M24	Actual	M24
Status	version 1.0		final ☐	
Nature	Report (R) ☒ Prototype (P) ☐ Demonstrator (D) ☐ Other (O) ☐			
Dissemination Level	Public (PU) ☒ Restricted to group (RE) ☐ Restricted to programme (PP) ☐ Consortium (CO) ☐			

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

Abstract (for dissemination)	This document discusses all the aspects of the design of a navigational benchmark, that is, a benchmark for testing small pattern matching queries that navigate around a small portion of the graph, starting from a given node. We enumerate the challenges these type of queries pose, what characteristics the data used by the benchmark needs to have, a set of chokepoints a query set proposal is architected around, and a set of performance metrics. It also includes query implementations for several vendor technologies.
Keywords	benchmark, choke points, dataset generator, graph database, query set, RDF, workload, auditing rules, publication rules, scale factors

Version Log			
Issue Date	Rev. No.	Author	Change
19/09/2014	0.1	Arnau Prat	First draft
28/09/2014	0.2	Arnau Prat	Reviewer' changes
28/09/2014	1.0	Arnau Prat	Final Version

Table of Contents

Executive Summary 4

Document Information 5

List of Figures 8

List of Tables 9

1 Deliverable Restructuring 11

2 Introduction 12

3 Challenges 15
3.1 Expressiveness . . . 15
3.2 Join pattern diversity . . . 16
3.3 Complexly structured data . . . 16
3.3.1 Cardinality estimation . . . 16
3.3.2 Data access patterns . . . 17
3.4 Parallelism exploitation . . . 17
3.5 Sort, unique and aggregates . . . 18

4 Chokepoints 19
4.1 Aggregation Performance. . . . 19
4.2 Join Performance. . . . 20
4.3 Data Access Locality. . . . 21
4.4 Expression Calculation. . . . 21
4.5 Correlated Sub-queries. . . . 22
4.6 Parallelism and Concurrency. . . . 22
4.7 RDF and Graph Specifics . . . 23

5 Data 24
5.1 Data Schema . . . 25
5.2 Scale Factors . . . 30
5.3 Distributions and Cardinalities . . . 31
5.3.1 Person-Knows-Person . . . 31
5.3.2 Message-hasCreator-Person . . . 32
5.3.3 Post/Comment-hasTag-Tag . . . 34
5.3.4 Person Name . . . 35
5.3.5 Post/Comment creationTime . . . 35
5.4 Determinism . . . 36
5.5 Time-Correlated entity id generation . . . 36
5.6 Update Streams . . . 37

6 Workload 39
6.1 Query Description Format . . . 39
6.2 Lookup Query Descriptions . . . 39
6.2.1 Query1 - Friends with a certain name . . . 39
6.2.2 Query2 - Recent posts and comments by your friends . . . 40
6.2.3 Query3 - Friends and friends of friends that have been to countries X and Y . . . 40
6.2.4 Query4 - New topics . . . 41

6.2.5 Query5 - New groups . . . 41
6.2.6 Query6 - Tag co-occurrence . . . 42
6.2.7 Query7 - Recent likes . . . 42
6.2.8 Query8 - Recent replies . . . 43
6.2.9 Query9 - Recent posts and comments by friends or friends of friends . . . 43
6.2.10 Query10 - Friend recommendation . . . 43
6.2.11 Query11 - Job referral . . . 44
6.2.12 Query12 - Expert search . . . 44
6.2.13 Query13 - Single shortest path . . . 45
6.2.14 Query14 - Weighted paths . . . 45
6.3 Update Query Descriptions . . . 46
6.3.1 Query1 - Add Person . . . 46
6.3.2 Query2 - Add Friendship . . . 46
6.3.3 Query3 - Add Forum . . . 46
6.3.4 Query4 - Add Forum Membership . . . 46
6.3.5 Query5 - Add Post . . . 46
6.3.6 Query6 - Add Like Post . . . 47
6.3.7 Query7 - Add Comment . . . 47
6.3.8 Query8 - Add Like Comment . . . 47
6.4 Choke-point coverage . . . 47
6.5 Substitution parameters . . . 48
6.6 Load Definition . . . 49

7 Performance Metrics 50
7.0.1 Recorded Metrics . . . 50

A Interactive Query Set Implementations 54
A.1 Virtuoso SPARQL 1.1 . . . 54
A.1.1 Query 1 . . . 54
A.1.2 Query 2 . . . 55
A.1.3 Query 3 . . . 55
A.1.4 Query 4 . . . 56
A.1.5 Query 5 . . . 56
A.1.6 Query 6 . . . 57
A.1.7 Query 7 . . . 57
A.1.8 Query 8 . . . 57
A.1.9 Query 9 . . . 58
A.1.10 Query 10 . . . 58
A.1.11 Query 11 . . . 59
A.1.12 Query 12 . . . 59
A.1.13 Query 13 . . . 59
A.1.14 Query 14 . . . 60
A.2 Virtuoso SQL . . . 61
A.2.1 Query 1 . . . 61
A.2.2 Query 2 . . . 62
A.2.3 Query 3 . . . 62
A.2.4 Query 4 . . . 63
A.2.5 Query 5 . . . 63
A.2.6 Query 6 . . . 64
A.2.7 Query 7 . . . 64
A.2.8 Query 8 . . . 64

A.2.9 Query 9 . . . 65
A.2.10 Query 10 . . . 65
A.2.11 Query 11 . . . 66
A.2.12 Query 12 . . . 66
A.2.13 Query 13 . . . 67
A.2.14 Query 14 . . . 67
A.3 Neo API . . . 68
A.4 Neo Cypher . . . 68
A.4.1 Query 1 . . . 68
A.4.2 Query 2 . . . 69
A.4.3 Query 3 . . . 69
A.4.4 Query 4 . . . 69
A.4.5 Query 5 . . . 70
A.4.6 Query 6 . . . 70
A.4.7 Query 7 . . . 70
A.4.8 Query 8 . . . 70
A.4.9 Query 9 . . . 71
A.4.10 Query 10 . . . 71
A.4.11 Query 11 . . . 71
A.4.12 Query 12 . . . 71
A.4.13 Query 13 . . . 72
A.4.14 Query 14 . . . 72
A.5 Sparksee API . . . 72
A.5.1 Query 1 . . . 72
A.5.2 Query 2 . . . 76
A.5.3 Query 3 . . . 77
A.5.4 Query 4 . . . 79
A.5.5 Query 5 . . . 80
A.5.6 Query 6 . . . 82
A.5.7 Query 7 . . . 83
A.5.8 Query 8 . . . 84
A.5.9 Query 9 . . . 86
A.5.10 Query 10 . . . 88
A.5.11 Query 11 . . . 90
A.5.12 Query 12 . . . 91
A.5.13 Query 13 . . . 93
A.5.14 Query 14 . . . 93

B Scale Factor Statistics 96
B.1 Scale Factor Statistics . . . 96
B.1.1 Scale Factor 1 . . . 96
B.1.2 Scale Factor 3 . . . 97
B.1.3 Scale Factor 10 . . . 98
B.1.4 Scale Factor 30 . . . 99
B.1.5 Scale Factor 100 . . . 100
B.1.6 Scale Factor 300 . . . 101
B.1.7 Scale Factor 1000 . . . 102

List of Figures

3.1 Target pattern of the triangle query. . . . 15
3.2 Triangle query expressed in SQL. . . . 15
3.3 Facebook graph's friendship cumulative degree distribution for all the world (global) and US as of May 2011 [27]. . . . 17

5.1 The LDBC-SNB data schema 26
5.2 A sample of the bucketed degree distribution. . . . 31
5.3 Average friends with respect to the number of persons. . . . 32
5.4 (a) Location of friends of Chinese people. (b) Location of friends of people from university A in China. . . . 33
5.5 (a) Number of posts per user for SF30. (b) Number of comments per user for SF30. . . . 34
5.6 Number of Posts, Comments and Friendship relations with respect to the scale. . . . 34
5.7 Distribution of the most popular names of Persons from Germany, Netherlands and Vietnam. . . 35
5.8 Flash-mob posts volume activity. . . . 36
5.9 Bit composition of Person Id. . . . 37
5.10 Bit composition of Message Id. . . . 37

List of Tables

5.1 Description of the data types. 25
5.2 Attributes of Forum entity. 27
5.3 Attributes of Message interface. 27
5.4 Attributes of Organization entity. 27
5.5 Attributes of Person entity. 28
5.6 Attributes of Place entity. 28
5.7 Attributes of Post entity. 28
5.8 Attributes of Tag entity. 28
5.9 Attributes of TagClass entity. 28
5.10 Description of the data relations. 30
5.11 Parameters of each scale factor. 30

6.1 Choke-point coverage 48
6.2 Interleaved latencies for each query type in miliseconds. 49

B.1 General statistics for SF 1 96
B.2 General statistics for SF 3 97
B.3 General statistics for SF 10 98
B.4 General statistics for SF 30 99
B.5 General statistics for SF 100 100
B.6 General statistics for SF 300 101
B.7 General statistics for SF 1000 102

1 Deliverable Restructuring

This deliverable details all the work done regarding Task 3.3. Originally, this task was projected to be presented by means of two separate deliverables, D3.3.3 and D3.3.4, called ‘Benchmark design for navigational benchmarking’ and ‘Benchmark design for pattern benchmarking’. For the following reasons, those deliverables have been merged into one.

Our goal was to make a separation between those queries expressed/expressible in imperative APIs (navigational) and those in declarative query languages (pattern). However, for multiple reasons we now think this does not make sense.

First, we have learned that navigational and pattern matching queries for the use cases targeted by LDBC-SNB, are slight variations of one another, not distinctly different query types. Conceptually, any navigational query (looking for paths, performing fixed/variable length traversals etc.) can be decomposed into a set of pattern matching queries plus aggregates: limit, group by, etc. As such, indexing techniques, query plan optimization techniques, execution engine optimizations, and posed challenges in general, apply to both types of queries interchangeably.

Second, as LDBC-SNB has no restrictions on the database technologies that can participate, there are many ways in which any of the queries can be implemented, therefore trying to separate queries based on the method of implementation is both limiting (it would not capture all implementations) and meaningless (it does not capture the nature of the queries themselves). Finally, many technologies provide APIs that are neither purely declarative nor purely imperative, and this is especially true when it comes to using their more complex language constructs, which are necessary for expressing graph queries.

We have instead settled on an alternate separation between the terms ‘navigational’ and ‘pattern matching’. Pattern matching queries are those queries that take as input a graph pattern and output a set of subgraphs that match that pattern. Then, navigational queries are a subset of pattern matching queries where the pattern has a specific starting point and 1 to N end points, thereby constraining the search space enough to make the query interactive. This way of separating the two is more useful. It communicates what a pattern matching query is, with clear graph-specific semantics. Further, it is based on the selectivity and complexity of queries, without requiring reference to how those queries are implemented.

2 Introduction

Data in multiple domains is typically modeled as graphs. With the explosion of the number of available data sources, an increasing amount of graph based applications has emerged. Some examples of graph applications are social networks, where both the users and vendors explore users' behavior in different ways, and protein-protein networks, where data is queried to find proteins with similar metabolic functions. These applications demand efficient systems to manage and query such data. Due to the variety of use case scenarios of graph data management systems, the number of choices is large; there exist a multitude of systems with different characteristics, targeting different types of workloads. Having benchmarks test and compare these systems under different situations is a great value-added for both users and vendors, as it allows the former to properly select the most suitable technology for their applications, and the latter to identify possible bottlenecks of their systems.

This deliverable describes the design of a benchmark to test the performance of graph data management systems for answering interactive navigational queries, which we define as a subset of pattern matching queries. Pattern matching queries take as input a graph pattern and return a set of subgraphs that match that pattern. A graph pattern is a combination of a graph motif and a set of predicates on the nodes and edges of the motif. The returned subgraphs can then be projected to produce a final result set consisting of, for example, a top-k set of entities. Graph pattern matching is an inherently complex problem, as subgraph isomorphism is NP-Complete [28]. Patterns can take many different shapes and forms. Depending on the complexity of the motif, and how restrictive the predicates are, the time to solve a graph pattern matching query might differ by orders of magnitude. Actually, the definition of a pattern is so flexible that any type of query could be potentially expressed as a combination of patterns, from simple queries to complex ones, as a single edge or a vertex match the definition of a pattern in its most basic form. Therefore, in this deliverable we are interested in narrowing the spectrum of pattern matching queries, by focusing into what we call interactive navigational queries and those systems designed to efficiently answer them. We define interactive navigational queries as those pattern matching queries where:

1. the pattern has a specific starting point and 1 to N end points;
2. the pattern has the form of a fixed number of hops or a variable-length path;
3. the pattern restricts (by means of predicates on vertexes and edges) the search space enough so the query can be answered interactively;

The term "navigational" comes from the nature of the queries, where one "navigates" from the starting point (i.e. a Person) to a connected region of the graph (a set of ending points). The benchmark presented in the present deliverable, is the foundation of the LDBC-SNB Interactive workload, whose specification draft is currently available online [16]. In the rest of this document we use the terms "interactive navigational queries" and "navigational queries" interchangeably.

Navigational queries are found in several domains, such as social networks, where people may ask for information about their friends and their activity, bibliographic systems, where users look for related publications to the topics they are interested in, or recommender systems, that look for similar products, movies, music etc., to cite just a few use cases.

The following is a list of simple navigational queries that could be potentially done in a social network site:

- Given a person, get the latest posts written by the friends of the person.
- Given a person, get all the friends of the person that have talked about a given topic in the last days.
- Given a person and a time interval, get all the friends of that person that share at least one friend.
- Given a person, get the groups that the friends of the person have recently joined.
- Given a person, get the likes to the person's posts after a given date.

- Given a person, get the shortest path from that person to another one.

Note that all these examples share the characteristic of having an input person, and require exploring the neighborhood of this person at a fixed or variable distance.

Although navigational queries look for simple patterns compared to other more complex queries that need to access a larger portion of the data, their execution still poses several interesting challenges. On the one hand, the inherently complex structure of graph data makes it usually difficult to exploit data locality, which can severely affect performance, especially in situations where data is stored on disk. Also, estimating cardinalities of intermediate sub-query results is difficult, as neighboring vertexes can have large overlaps in their adjacencies, and cardinality distributions are highly skewed. Finally, exploiting parallelism, either data driven or task driven, is an arduous task often resulting in load imbalances and inefficient use of resources. On the other hand, although expressing navigational queries using standard relational query languages (i.e. SQL) is possible, graph semantics are lost in the process hence losing the opportunity to apply graph oriented optimizations. Finally, as each graph traversal step is conceptually a join, navigational queries' execution plans may lead to complex plans with multiple joins, sometimes difficult to optimize. These challenges are described in more detail in Chapter 3.

Navigational queries are look-up queries by nature, but graph applications are usually dynamic systems that new data is periodically fed into. A navigational benchmark must also contain updates, as these might have important implications on how the graph management systems are internally architected.

Broadly speaking, a navigational query benchmark should consist of the following parts:

1. data with a similar structure as that found in real datasets (with comparable distributions, cardinalities and correlations), as many of the challenges posed by pattern matching queries and navigational queries by extent highly depend on the data queried;

2. a set of representative navigational queries with patterns similar to those observed in real interactive graph applications, which can stress both the expressiveness of the used query languages and test the effectiveness of the query optimizers and runtime engines;

3. a set of update queries;

4. a workload consisting of a combination of the designed lookup and update queries and a set of substitution parameters;

5. a selection of representative performance metrics taking into account the interactive nature of the navigational benchmark's target systems;

These parts are architected around a set of choke-points, that is, a set of identified aspects the benchmark wants to specially test: expressiveness of their query languages or APIs, efficiency of their query optimizers in the case of having a declarative language and query execution engines, the ability to parallelize the queries, etc. From these choke-points, a realistic and challenging test bed is derived, which, along with a set of comprehensive performance metrics, a representative framework to compare graph management systems for navigational queries is defined.

There have been multiple attempts in the literature to propose benchmarks for graph management systems. Renzo et al. proposed a micro-benchmark based on small micro-queries performed on top of a social network [1]. In their approach the authors divided the queries into several categories, based on their complexity, but they are in in general very simple and mostly limited to two hops. Queries proposed in LDBC-SNB interactive workload are more complex and based not only on real use cases but also on real choke-points. Furthermore, data used in that work is not as realistic and rich as that used in LDBC-SNB, being limited to reproduce realistic distributions based on the RMAT model [5]. Finally, that proposal does not contain updates.

Other benchmarks are more focused on specific domains, such as those of community detection [3] or graph traversal operations [6]. In [18], Macko et al. propose PIG, a set of operations to test the performance of graph databases. These operations include small micro operations, graph algorithms, and traversals. They use both synthetic and real graphs. The main drawbacks of PIG are, first, that queries do not look representative of actual use cases of graph databases, that combine both traversals and attribute lookups and, second, that databases

are accessed via a common third party programmatic API, rather than their native APIs, limiting their ability to perform query optimizations. A similar drawback is found in the benchmark proposed by Pobiedina et al. [22], which look for randomly generated patterns of several sizes. Finally, Facebook presented Linkbench [2], a benchmark centered on the OLTP workload on the Facebook graph. Nevertheless, Linkbench uses synthetic graphs that do not reproduce the real structure of those found in real networks, besides the degree distributions. The authors argue that the graph does not significantly affect the performance of the type of queries Facebook perform, something not shared with LDBC-SNB.

In general, existing proposals do not follow a choke-point and use case centered design, but focus on isolated micro operations or domain centered algorithms. Moreover, data used do not contain realistic correlations, which can significantly affect the performance of the queries. These issues are both addressed by LDBC-SNB benchmarks, being the interactive workload presented in this deliverable that focus on more simple navigational queries.

The rest of the document is structured as follows. In Chapter 3 we describe in depth the different challenges posed by navigational queries. In Chapter 4 we introduce the choke-points around which the benchmark is architected; in Chapter 5 we describe and argue the characteristics of the data and in Chapter 6 we describe the queries used in the benchmark, and define the performance metrics. Finally, in Appendix A we show implementations of the proposed queries for existing languages or APIs, and in Appendix B we show the statistics of the used datasets.

3 Challenges

3.1 Expressiveness

One of the most relevant challenges posed by navigational and pattern matching queries in general, is how do we express them. Although traditional relational model query languages (i.e SQL) are not designed to express graph patterns, these can usually still be expressed since, conceptually, each navigational step in a graph pattern can be translated to a join operation. However, expressing these types of, commonly deeply nested, queries is complex and graph semantics are often lost in the process. Furthermore, some queries comprise of recursive patterns, which sometimes requires the use of proprietary extensions that might not be cross-platform, or even the use of embedded procedural languages. In such a situation, the ability of the query optimizer to perform graph oriented optimizations is highly hindered. For example, one might think about the following query:

> *"Given a person, find those friends of that person that close at least one triangle with him."*

Figure 3.1 shows the pattern this query is looking for. One straightforward structural optimization a query optimizer could do, is to early prune those neighbors of the input node with a degree of one, since they cannot close any triangle. Also, if one had structural properties precomputed, those nodes or edges with a clustering coefficient of 0 could also be discarded.

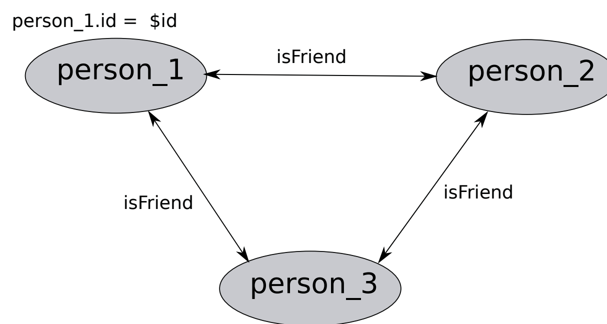


Figure 3.1: Target pattern of the triangle query.

Figure 3.2 shows the same query implemented in SQL. Two main conclusions can be extracted from this implementation. The first is the presence, even with this simple pattern, of three joins between the “isFriend” relation, which exemplifies the complexity of graph pattern matching queries. The second is that the graph notion is lost and it is very difficult for the optimizer to apply optimizations based on the structure.

```
select  k3.id1
from isFriend k1, isFriend k2, isFriend k3
where
    k1.id1 = @id@ and
    k1.id2 = k2.id1 and
    k2.id2 = k3.id1 and
    k3.id2 = @id@
```

Figure 3.2: Triangle query expressed in SQL.

Expressing graph patterns declaratively requires for higher level languages where the graph structure can be explicitly expressed, including recursive patterns. Several attempts to define a comprehensive, flexible and powerful query language have been performed by both academia and industry such as SPARQL [9], G-SPARQL [24], Snql [25], Graphql [10], Cypher [26], Infinite graph language [11] or Gremlin [8], but a standard graph query

language is yet to be defined. For a detailed description on query languages and their characteristics, see Deliverable 3.3.2 [15].

In this scenario, some existing graph database technologies propose the use of APIs to query graphs. These kind of languages are highly expressive and can implement arbitrary algorithms. A highly skilled programmer, with the proper knowledge about the data and their characteristics (i.e. distributions and cardinalities), may be able to implement highly optimized versions of the queries. However, having developers with such skills is not always possible. Additionally, APIs often require intermediary results to be materialized and serialized into objects, which are then further processed by the implementation. This materialization is expensive and can be avoided in systems with declarative query languages, which are more aware of the query's intent and therefore better equipped to perform performance optimizations, e.g., performing predicate checks as close to the data as possible.

3.2 Join pattern diversity

Besides the simpler navigational queries such as "Getting the friends of a specific person" or "Getting the posts made by a specific person", where only data within one hop is accessed, typically navigational queries imply navigating the graph through paths of variable length, comprised of multiple relationship types that, in turn, connect entities of different types. As each navigational step is conceptually a join, these queries are translated to diverse query plans, consisting of multiple joins with operands of diverse cardinalities (which are sometimes difficult to estimate, as we will explain shortly). This situation leads to diverse execution plans, even for the same query, depending on the substitution parameters. This is a challenging environment for the optimizer, which tests its ability to produce good execution plans, by properly selecting join operators, access methods and their implementations, pushing down selections and delaying projections.

3.3 Complexly structured data

Many of the challenges posed by navigational queries directly or indirectly depend on the particular characteristics of the data queried in graph applications. These are described in the following sections.

3.3.1 Cardinality estimation

Correctly estimating the cardinality of intermediate query results is crucial when estimating the cost of the different execution plans considered during the query optimization process. While cardinality estimation is easy when attribute values of a relation are distributed uniformly, this is not the case when distributions are skewed. Skewness is present in real data typically used by graph applications. For example, in social networks, the number of friends of persons typically follows skewed distributions such as power laws. As a consequence, we have most of the people having a few friends, while few people have a lot of friends. In Figure 3.3, we show the cumulative degree distribution of the Facebook graph of all the world and US in May 2011. The y axis shows the fraction of friends having a degree of k or more. We see that the majority of users has a moderate number of friends, less than 200, while the number of people with a large number of friends is small.

Another characteristic that complicates the estimation of cardinalities is the highly clustered nature of real graphs (i.e. they contain a relatively large number of triangles compared to other types of networks). This implies that neighborhoods of vertexes may have large overlaps, which complicates the estimation of cardinalities in the presence of joins plus "distinct" clauses. For example, imagine a query that implies accessing the distinct friends of the friends of a person in a social network. Many of the person's friends can share friends, and therefore lots of duplicates may be present.

These irregularities pose additional complications when optimizing graph queries in general, either for query optimizers that may need to produce different execution plans for the same query for different execution parameters, or when these are implemented using imperative languages.

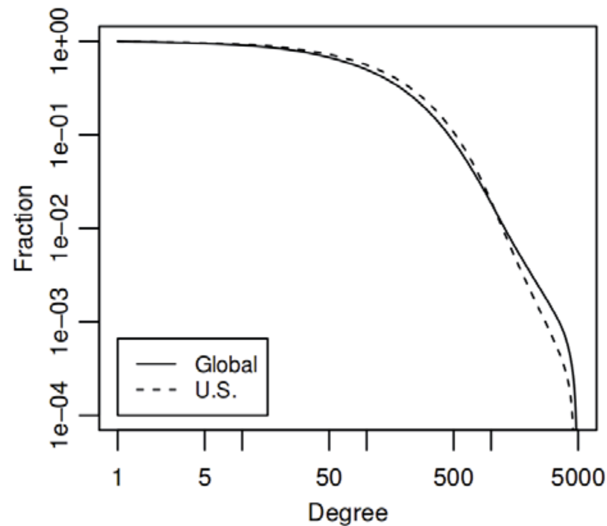


Figure 3.3: Facebook graph's friendship cumulative degree distribution for all the world (global) and US as of May 2011 [27].

3.3.2 Data access patterns

The complex nature of data in graph applications has as a consequence that entities usually accessed together are not stored together. This makes data accesses non local, with the implications this has in performance, which may specially affect when data is stored in external storage. Poor locality is specially manifested when looking for patterns that go beyond the first hop of adjacencies. Although data can be laid out so immediate neighbors of a vertex are placed in a more local way by exploiting the clustered nature of real data [23], as we increase the number of hops, access patterns become more and more random.

3.4 Parallelism exploitation

Parallelism can appear at different levels of granularity:

- inter-query level: different queries are executed in parallel;
- intra-query level: different "branches" or groups of operations in the execution plan are executed in parallel;
- operator level: different threads are spawned for a given operator to be executed in parallel.

The most suitable granularity depends on the actual workload to be executed. One may argue that, since navigational queries touch a small amount of data and are simple enough to be solved interactively, so it is not worth to parallelize queries at the intra-query or operator levels, as spawning threads with little work to do could become counter productive. However, exploiting query level parallelism can be cache inefficient, as data from one query can evict from the cache data needed by other queries.

Since graph data is complexly structured, as we have previously discussed, there are situations where it is possible to exploit parallelism at the intra-query or even at the operator levels, more specifically when data accessed by the query is large enough to be worth to spawn multiple threads to process it. At these levels of parallelism, cache is more efficiently used, as data used by the different threads can remain in the caches and be reused by other threads, even executing other parts of the execution plan.

To sum up, depending on the actual workload, the query optimizer (if any) has to decide at which level it will exploit parallelism to maximize the performance. In the absence of a query optimizer, exploiting parallelism can become an arduous task, and therefore it is often limited to inter-query level parallelism.

3.5 Sort, unique and aggregates

One common characteristic of navigational query applications is that only the top-k results are presented to the end user. This implies sorting results by some criterion which sometimes depends on some sort of aggregates. Finally, due to the clustered nature of the data queried, it is frequent to have repeated results, and these need to be removed so only unique results are presented. These features add even more complexity to navigational queries.

4 Chokepoints

In Chapter 3 we have described the particular characteristics and challenges that makes navigational queries special. Besides these challenges, many other aspects of database systems play a significant role when executing navigational queries and other types of workloads. In this section we create a list of "choke points", that is, low level aspects (some of them influenced by the nature of navigational queries), that queries should test in order to make the benchmark challenging and interesting enough. Some of these choke-points are based on those already identified in TPC-H [4].

We define seven categories of choke-points, each of them containing a different number of aspects to test. These categories are: Aggregation Performance, Join Performance, Data Access Locality, Expression Calculation, Correlated Sub-queries, Parallelism and Concurrency, and RDF and Graph Specifics. Each category contains a set of more specific sub-choke-points, which can be classified depending on the part of the database system they affect:

- QOPT: Those aimed at testing aspects of the query optimizer.
- QEXE: Those aimed at testing aspects of the execution engine.
- STORAGE: Those aimed at testing aspects of the storage system.

For the sake of understanding, choke-points are expressed mostly in terms of relational algebra, but they are applicable to non-relational systems.

4.1 Aggregation Performance.

Navigational queries generally have a *top-k* order by and often a *group by* in addition to this, as users are only interested in the most relevant results. The presence of *aggregates*, *group* by and *distinct* offers multiple optimization opportunities for either the query optimizer or the query programmer. Another feature queries have are distinct operators, i.e. distinct friends within two steps. Collectively these are all set operations that may be implemented with some combination of hash and sorting, possibly exploiting ordering in the data itself. Finally, we have complex aggregates that go beyond the most basic count and sums. All these features allow for optimizations in both the query optimizer and the execution engine. The following are the choke-points related to aggregates:

- **CP1.1 QOPT**: Interesting orders.
  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.

- **CP1.2 QEXE**: High cardinality *group-by* performance.
  This choke-point tests the ability of the execution engine to parallelize *group-by's* with a large numebr 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.

- **CP1.3 QEXE**: Complex aggregate performance.
This choke-point test the performance of the execution engine to perform complex aggregates. Many databases offer user defined aggregates and more complex aggregation operations than the basic *count*, *sum*, *max* and *min*, for example string concatenation aggregation operator. These types of aggregates are expected to benefit from the same optimizations as the basic built-in ones, for example partitioning.

- **CP1.4 QOPT**: 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.

- **CP1.5 QEXE**: Dependant group-by Keys.
This choke-point tests the ability of the query optimizer to exclude those functionally dependant *group-bys*. Sometimes queries require for group-by's on a set of columns and a key, where the value of the key determines the columns. In this situation, the aggregation operator should be able to exclude certain group-by attributes from the key matching.

4.2 Join Performance.

As explained in Chapter 3, each graph traversal step is conceptually join. In navigational queries, one navigates from a starting point to one or more endopoints through different edge types, which may have different cardinalities and distributions. Therefore, join patterns are diverse. Properly selecting the join operator performance, or the order in which the joins (the traversals) are performed may have a great impact on the performance. Queries are designed so as to reward judicious use of hash join by having patterns starting with one entity, fetching many related entities and then testing how these are related to a third entity, e.g. posts of a user with a tag of a given type.

- **CP2.1 QOPT**: Rich join order optimization.
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 reorderable 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.

- **CP2.2 QOPT**: Late Projection.
This choke-point tests the ability of the query optimizer to delay the projection of uneeded 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 ommit 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.

- **CP2.3 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 appropiate. 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.

- **CP2.4 QEXE**: Sparse Foreign Key Joins.
  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 [7].

4.3 Data Access Locality.

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

- **CP3.1 QOPT**: Detecting Correlation.
  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.
- **CP3.2 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.
- **CP3.3 QEXE**: Scattered indexed access pattern.
  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.

4.4 Expression Calculation.

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

- **CP4.1 QOPT**: Common subexpression elimination (CSE).
  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.

- **CP4.2 QOPT**: Complex boolean expression joins and selections.
  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 [19].

4.5 Correlated Sub-queries.

Queries have many correlated sub-queries, for example constructs like x within two steps but not in one step, which would typically be a correlated sub-query with NOT EXISTS. There are also scalar sub-queries with aggregation, for example returning the count of posts satisfying a certain criteria.

- **CP5.1 QOPT**: Flattening sub-queries.
  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.

- **CP5.2 QEXE**: Overlap between outer and sub-query.
  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 [21] 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.

- **CP5.3 QEXE**: Intra-query result re-use.
  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.

4.6 Parallelism and Concurrency.

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

- **CP6.1 QEXE**: Inter-query result re-use.
  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.

4.7 RDF and Graph Specifics

Estimating cardinalities in graph data is particularly hard. For example, a query optimizer needs to recognize whether a relationship has a tree or graph shape in order to make correct cardinality estimations. Further, there are problems aggregating properties over a set of consecutive edges. Queries contain business questions dealing with paths and aggregates across paths, as well as the easier case of determining a membership in a hierarchy with a transitive part-of relation.

- **CP7.1 QOPT**: Translation of internal ids into external ones.
  This choke-point tests the ability of the query optimizer to delay the translation between internal and external entity ids to late in the query. Translate at point of minimum cardinality, e.g. after top k order by RDF and possibly graph models often use a synthetic integer identifier for entities, e.g. URI's . For presentation to the client applications, these identifiers must be translated to their original form, e.g. the URI string that was used when loading the data. This should be done as late as possible, or at the point of minimal cardinality in the plan.

- **CP7.2 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.

- **CP7.3 QEXE**: Execution of a transitive step.
  This choke-point tests the ability of the query execution engine to efficinently execute transitive steps. Graph workloads may have transitive operations, for example finding a shortest path between vertices. 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 vertex 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 vertices. There are many possible tradeoffs between generality and performance.

5 Data

Data is one of the most important aspects of a navigational benchmark. Many of the challenges posed by navigational queries directly or indirectly depend on the data. Data used in a navigational benchmark should capture the main characteristics of data used in real applications. This includes distributions, cardinalities and correlated edges between similar entities, this last one producing clustered graphs. Furthermore, data must be rich enough (containing a variety of different entities and attributes) to allow navigational queries testing all the choke points introduced in Chapter 4.

One may think of using real data for a navigational benchmark, but this has several drawbacks that makes this option not always the best choice. First of all, a benchmark must contain data of different scales, to test systems of different sizes and characteristics. Obtaining real data with the size needed by the benchmark might not be always possible, either for technical reasons or for privacy concerns. Second, characteristics of a single real dataset might not fit all the needs of the data needed by the benchmark (distributions, cardinalities, attribute diversity).

For all these reasons, it is sound to think about generating synthetic data, as it provides the flexibility and the control over what features data need to have. In Deliverable 2.2.2 [12] we described the design of DATAGEN, a data generator used to produce synthetic social networks with characteristics similar to those found in the real world. In the present deliverable, we complement that description with new added features that directly affect the design of a navigational benchmark. Also, for the sake of completeness, we describe the schema of the data produced again (significant updates were performed since the publication of Deliverable 2.2.2). The list of major changes in the data generator with respect to that described in Deliverable 2.2.2 is the following:

- **Updates on the data schema**: The data schema has been updated to support new relations, such as adding tags to comments as well as dates to the Knows relationships.
- **Updates on edge and data distributions**: Several distributions have been adjusted, including a new friendship distribution resembling to that found in Facebook and the inclusion of time correlated posts and comments.
- **Determinism**: The implementation of the data generator has been adapted to produce deterministic results regardless of the number of machines used and the platform.
- **Time-Correlated id generation**: The ids of generated entities are now correlated with the entity creation time, as one would expect to see in a real system. This is important to make the data loading times and data memory layouts more representative to those found in real applications.
- **Update streams**: The data generator has been extended to produce the update transaction streams needed for the navigational benchmark.

5.1 Data Schema

Table 5.1 describes the different types used in the whole schema.

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

Table 5.1: Description of the data types.

Figure 5.1 shows the data schema in UML. The schema defines the structure of the data used in the benchmark in terms of entities and their relations. Data represents a snapshot of the activity of a social network during a period of time. Data includes entities such as Persons, Organizations, and Places. The schema also models the way persons interact, by means of the friendship relations established with other persons, and the sharing of content such as messages (both textual and images), replies to messages and likes to messages. People form groups to talk about specific topics, which are represented as tags.

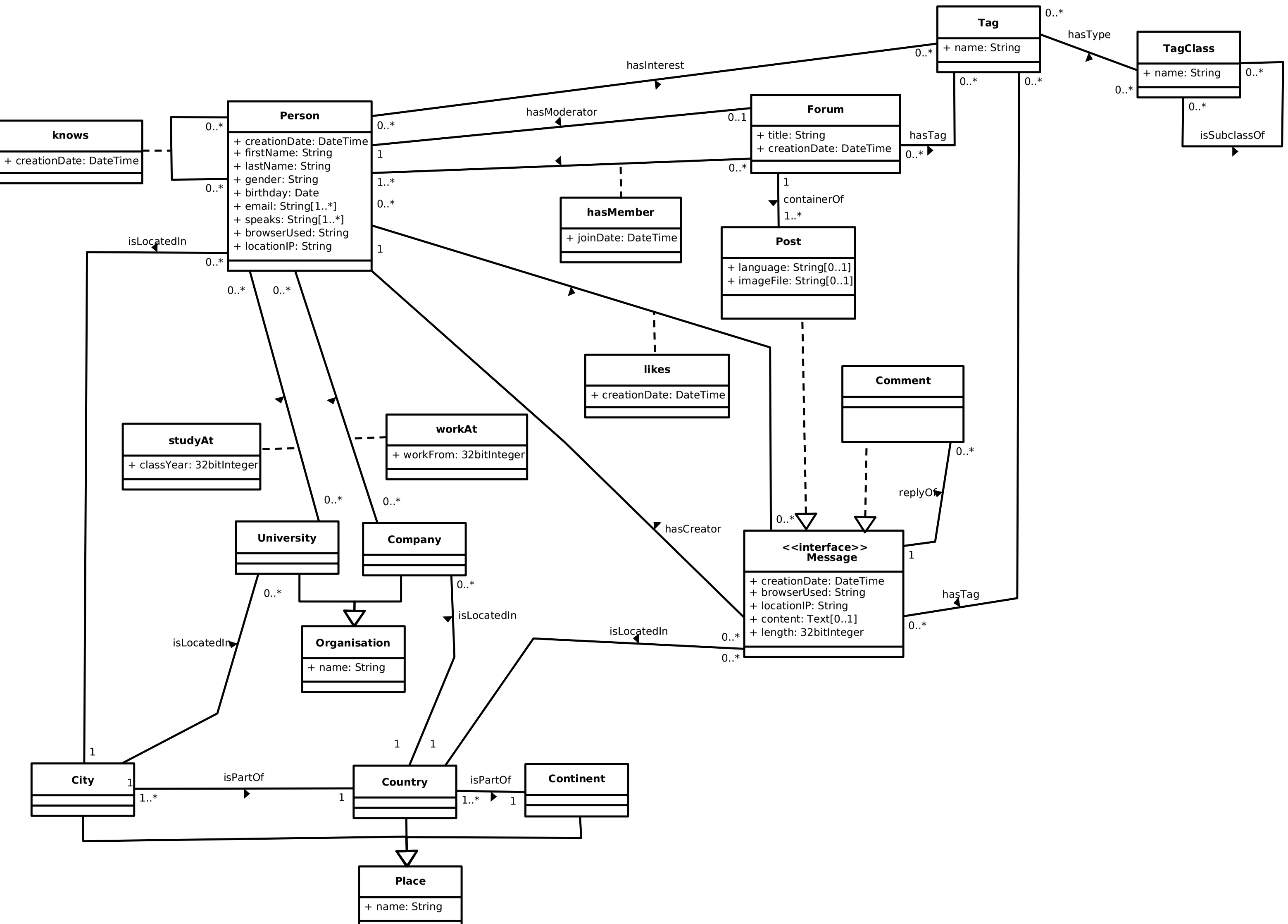


Figure 5.1: The LDBC-SNB data schema

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

Entities

City: 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 Organization, 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 5.2 shows the attributes of Forum entity.

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

Table 5.2: Attributes of Forum entity.

Message: an abstract entity that represents a message created by a person. Table 5.3 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	32bitInteger	The length of the content.

Table 5.3: Attributes of Message interface.

Organization: an institution of the real world. Table 5.4 shows the attributes of Organization entity.

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

Table 5.4: Attributes of Organization 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 5.5 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	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 5.5: Attributes of Person entity.

Place: a place in the world. Table 5.6 shows the attributes of Place entity.

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

Table 5.6: 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 5.7 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 5.7: 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 5.8 shows the attributes of Tag entity.

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

Table 5.8: Attributes of Tag entity.

TagClass: a class or a category used to build a hierarchy of tags. Table 5.9 shows the attributes of TagClass entity.

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

Table 5.9: Attributes of TagClass entity.

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

Relations

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

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

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

Table 5.10: Description of the data relations.

5.2 Scale Factors

One of the advantages of having a synthetic data generator, is the ability to generate data of different sizes. When designing a benchmark, several scale factors targeting systems of different sizes have to be defined. In the case of DATAGEN, these scale factors (SFs) are computed based on the ASCII size in Gigabytes of the generated output files using the CSV serializer. For example, SF 1 weights roughly 1GB in CSV format, SF 3 weights roughly 3GB and so on and so forth. The proposed SFs are the following: 1, 3, 10, 30, 100, 300, 1000.

The size of the resulting dataset, is mainly affected by the following configuration parameters: the number of persons and the number of years simulated. Different SFs are computed by scaling the number of Persons in the network, while fixing the number of simulation years. Table 5.11 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 5.11: 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. Choosing the dataset size to define the scale factors is important for the following reasons. On the one hand, cost estimation functions of execution plans usually use I/O operations to estimate the cost of executing a query, as the disc is typically the bottleneck (sometimes the network) of a database system and the disc size limits the amount of data able to be processed by the system. On the other hand, disc size is a structural agnostic metric, that is, it allows to compare data regardless of the structure of the data itself. If we used some structural characteristic, such as the number of persons in a network, as distributions and cardinalities of different datasets might differ, the computational power needed to process two datasets with the same number of nodes might differ significantly.

In Appendix B.1 we show the size in bytes for the different scale factors, as well as the memory spent for each of the entities/relations.

```
0 1
0 1
0 1.5
1.5 2.2
2.2 3.55
3.55 4.37
4.37 5.37
5.37 6.61
6.61 8.13
...
570 623
623 674
674 723
723 781
781 863
863 1029
1029 5000
```

Figure 5.2: A sample of the bucketed degree distribution.

5.3 Distributions and Cardinalities

Distributions and cardinalities play a significant role on queries' complexity. In such real data, distributions of edges and attributes are typically skewed, following power-law-like shapes, and they often exhibit correlated values. This poses several challenges to solving navigational queries, as explained in Chapter 3. When designing a navigational benchmark, synthetically generated data needs to reproduce those distributions found in real data. In the following sections we describe the most relevant distributions, concerning the queries described in Chapter 6.

5.3.1 Person-Knows-Person

In the case of a social network, one of the most important distributions is probably the Friendship distribution, as typical navigational queries performed by a user on such data implies navigating through the friendship relationships of the input Person. On the one hand, the cardinality of the friendship relationships directly affects the amount of data the query will have to access. If the average friendship degree is too small, navigational queries will not touch enough data to be challenging enough. On the other hand, if it is too large, queries will stop to be interactive and therefore, stop being representative for a navigational benchmark. Furthermore, the amount of friends must not be uniform among all the persons in the network, but highly skewed, complicating the work of the query optimizer on estimating the cardinalities of intermediate results. .

For the sake of realism, DATAGEN reproduces the Facebook's distribution shown in Figure 3.3, which contains the cumulative distribution of the friendship degree of the Facebook graph as of May 2011. As there is not a mathematical model for this distribution, DATAGEN opts to sample it. From the Facebook distribution, DATAGEN discretizes the distribution into 100 equi-width buckets (all buckets contain 1% of the Persons of the graph). Figure 5.3.1 shows a sample of the file containing the distribution, where the the first and second columns are where the left and the right extremes of the buckets fall respectively. Once we want to extract the degree of a person from the distribution, we pick up a random bucket uniformly distributed, and select a degree within the range of the bucket uniformly distributed as well.

Nevertheless, we cannot directly use the distribution as is, as the size of the network DATAGEN generates might not be as large as the Facebook graph i.e. it is not sound to have Persons with 5k friends in a 10K network. For this reason, DATAGEN scales down the Facebook distribution to match a desired average degree. According

to [27], the Facebook graph average degree is 190 ($fbavg$). Therefore, given a target average degree $tavg$, we can compute the new ranges of a given bucket as follows:

$$min'_i = \frac{min_i \cdot tavg}{fbavg}$$
$$max'_i = \frac{max_i \cdot tavg}{fbavg}$$

where min_i and max_i are the min and max of the bucket i.

To compute $tavg$, DATAGEN uses the following equation:

$$tavg = s^{0.512-0.028 \cdot log(s)}$$

where s is the size of the network. This formula produces different average degrees for different network sizes, increasing logarithmically. The average degrees are chosen to be a tradeoff between being realistic, and allowing the test of small systems at the same time. The average degree per scale factor is shown in Figure 5.3, which scales sub-linearly.

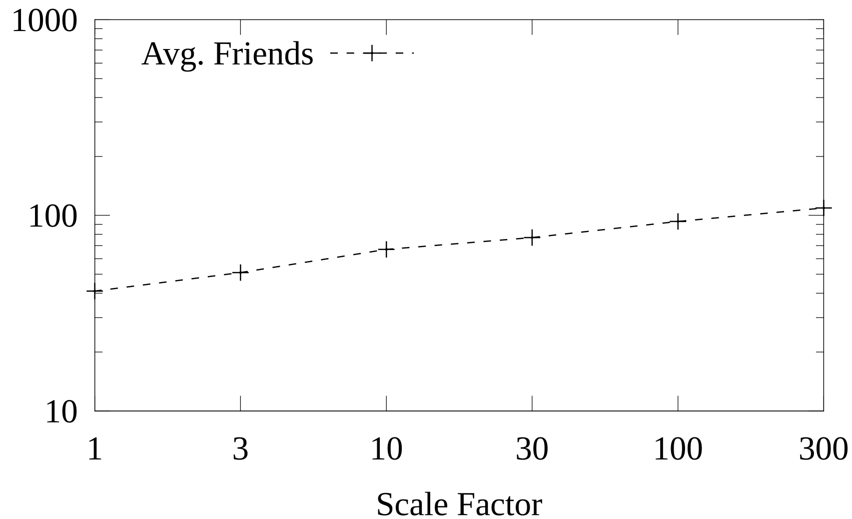
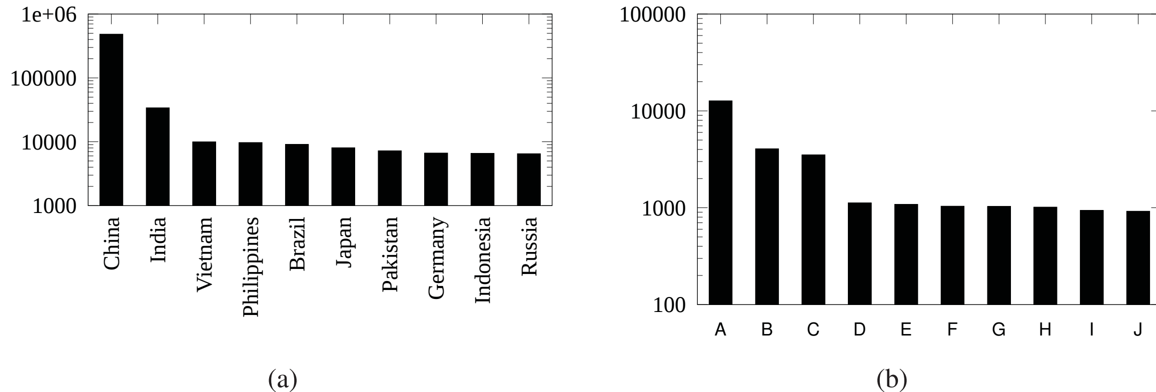


Figure 5.3: Average friends with respect to the number of persons.

Besides the friends degree distribution and average friends, another important feature needs to be considered: correlated friends. As described in Deliverable 2.2.2, DATAGEN produces correlated edges, that is, people with similar characteristics tend to be connected. This is exemplified in Figure 5.4, where we show (a) the location of the friends of Persons from China, and (b) the location of friends from persons in China studying in University A (names has been simplified due to space restrictions). We see that Persons from China have mostly friends from China, and the rest of friends from other countries belong to countries geographically close to China. On the other hand, friends of Persons studying in A, study also mostly in A, and the rest of universities (B, C, D, etc...) are also Chinese universities. In Chapter 3 we described the importance of this feature regarding the complexity of navigational queries.

5.3.2 Message-hasCreator-Person

Besides the friendship relationships, many navigational queries imply accessing data concerning the activity of the Persons in the network, specially that from the friends of a given Person. This activity is represented in our schema as Posts, Comments and Likes to Messages.



(a) (b)

Figure 5.4: (a) Location of friends of Chinese people. (b) Location of friends of people from university A in China.

In a real social network, not every Person has the same level of activity inside the network. Some people are very active, so they post frequently while others post more sporadically. It is sound to think that there is a correlation between the number of friends of a person and their level of activity, so people more active tend to have more friends and vice versa. Conceptually, DATAGEN produces two types of Forums: those commonly known as the “Wall” in the Facebook’s nomenclature, were only the owner of the Forum can create Posts on it, which can be replied only by their friends, and “Groups”, which are sets of people that gather together to talk about a specific topic. In the schema, no distinction is made between them. In DATAGEN, we correlate Posts a user creates in its “Wall” with the number of friends she has, using the following formula:

$$posts = (posts_per_month \cdot months) \cdot \frac{friends}{max_friends}$$

where $posts_per_month$ is the number of Posts per month a Person can create in their Forum, $months$ is the number of months from the Person creation date to the end of the simulation time, $friends$ is the number of friends of the user and $max_friends$ is the maximum number of friends a Person can have.

DATAGEN also correlates the number of Posts created by Persons in a “Group”, with the number of friends, though indirectly. Given a “Group”, the number of Posts made by Persons on that “Group” is computed with the following formula:

$$posts = (group_posts_per_month \cdot months) \cdot \frac{members}{max_members}$$

where $group_posts_per_month$ are the number of Posts per month can be created by people in the “Group”, $months$ is the number of months from the “Group” creation date to the end of the simulation time, $members$ is the number of members of the “Group” and $max_members$ is the maximum number of members a “Group” can have.

Since in DATAGEN the number of “Groups” a Person belongs to depends on the number of friends the Person has, then the number of Posts of Persons in Forums also depend on the number of friends.

The number of Comments a Post have is a random number between 1 and $max_comments$. The author of a reply to a Post/Comment is selected within the list of friends of the Post’s author, in the case of a Post from a Persons’ Forum or the list of members in the case of a Post from a public Forum. Therefore, the larger the number of friends, the larger the probability to be selected to reply to a Post. Figure 5.5 shows the distributions of both Post and Comments for SF30. We see that the distribution is skewed, as expected considering we are using a skewed friendship distribution.

Finally, note that the level of user activity scales with the size of the network, as does the average degree. Figure 5.6, shows the total number of Posts and Comments with respect to the scale factor. We see that the larger

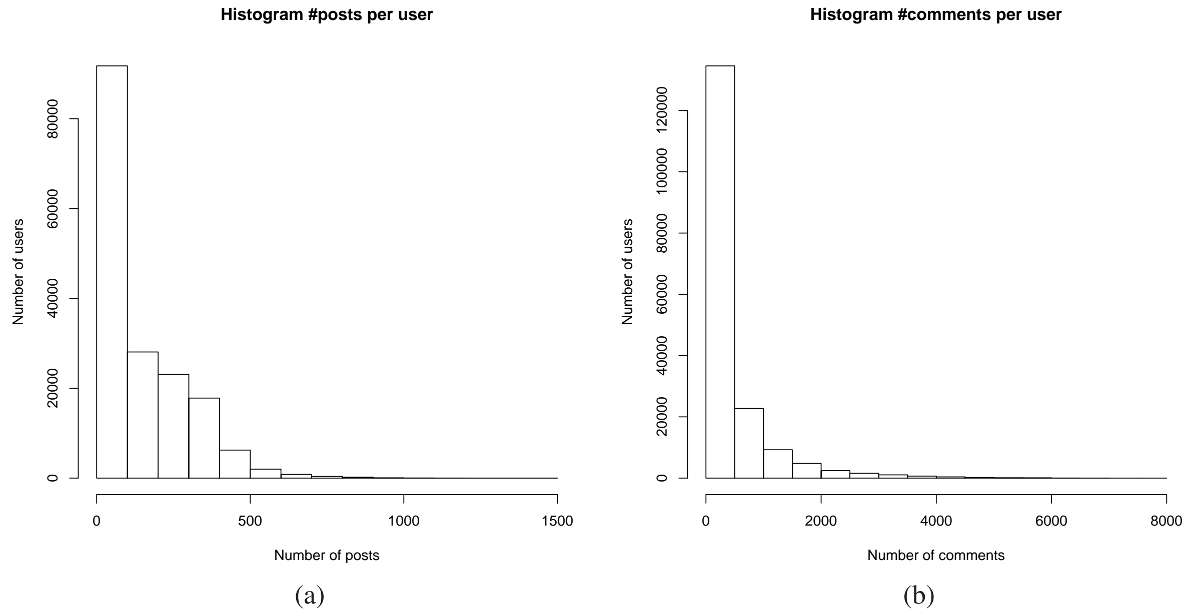


Figure 5.5: (a) Number of posts per user for SF30. (b) Number of comments per user for SF30.

the scale factor, the larger the number of Posts and Comments. If we approximate the monomial trend line for comments, we observe that the number of comments scales close to linearly with respect to the scale factor, which is something expected as comments are the entities that impact the most on the disk size of the datasets, as shown in Appendix B.1.

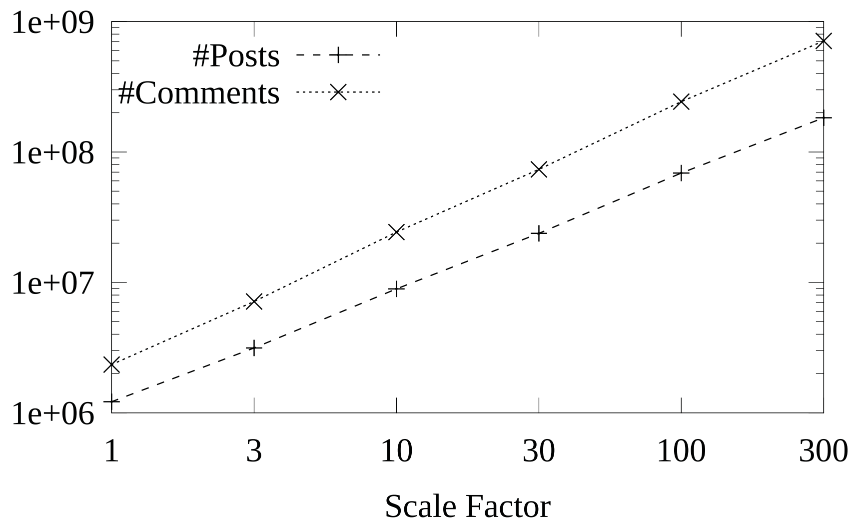


Figure 5.6: Number of Posts, Comments and Friendship relations with respect to the scale.

5.3.3 Post/Comment-hasTag-Tag

Another important characteristic worth considering is the topic of Posts and Comments. As Persons connected to other Persons tend to have similar interests, it is sound to think that they speak about similar topics. How this might affect the execution of a query can be seen in the following example. In a query needing to retrieve the

topics the friends of a Person are talking about, the amount of different topics will probably be smaller than if the topics were chosen randomly. In DATAGEN, Posts and Comments created by Persons have a higher probability to talk about Persons' interests. Similarly, Persons tend to join groups talking about topics they are interested in.

5.3.4 Person Name

It is known that people's names are not distributed uniformly. Some names are more popular than others, and this popularity is heavily affected by the nationality of the person. DATAGEN uses dictionaries extracted from DBpedia to reproduce this behavior. For each country, there is a subset of names with a large probability to appear (following a power-law distribution), while the rest of names are considered to appear uniformly. Figure 5.7, shows the distribution of the most popular names of Persons whose location is Germany, Netherlands and Vietnam for a dataset of SF10. We see that there are names more popular than others, and that this popularity depends on the country persons are located in.

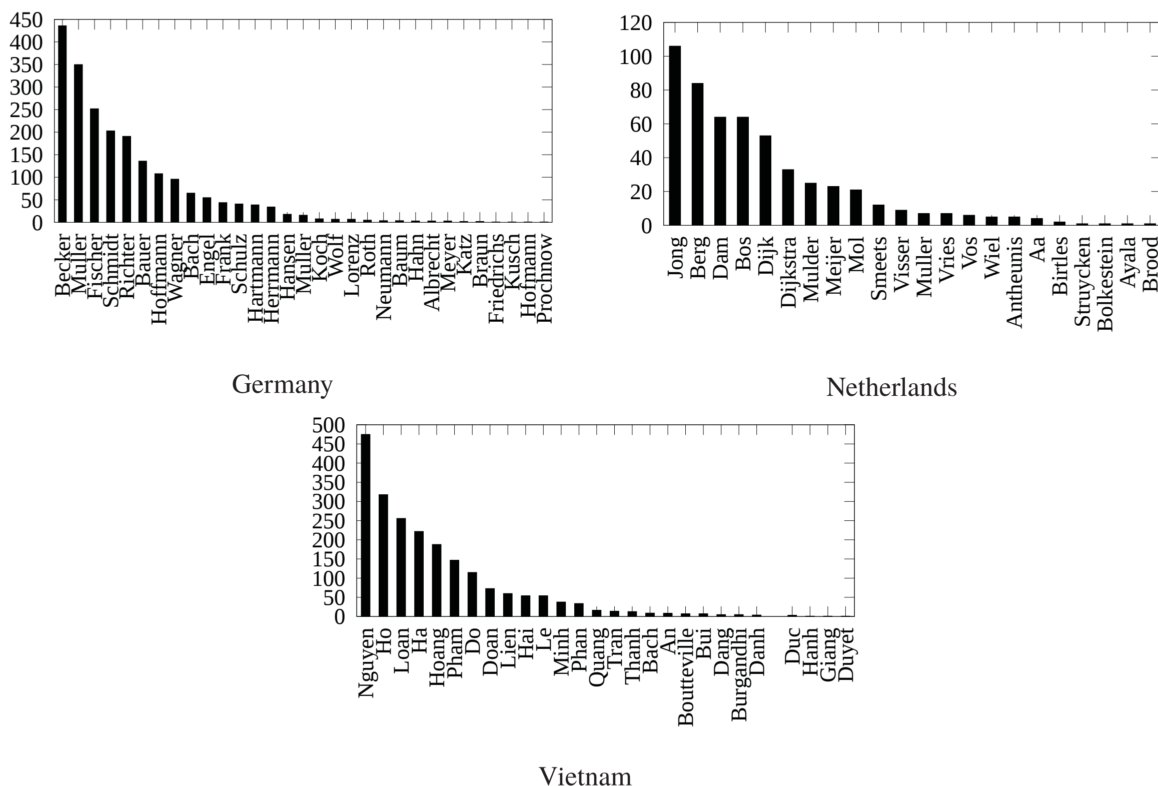


Figure 5.7: Distribution of the most popular names of Persons from Germany, Netherlands and Vietnam.

5.3.5 Post/Comment creationTime

User activity is not uniform, and it is typically driven by real world events: Olympic games, elections, natural disasters, etc.. When these events happen, user activity concerning the event increases, especially from those users interested in the event.

DATAGEN reproduces this behavior with the simulation of what we name as flash-mob events. Several events are generated randomly at the beginning of the data generation process, which are assigned a random tag, and are given a time and an intensity which represents the repercussion of the event in the real world. When persons' posts are created, some of them are classified as flash-mob posts and their tags and dates are assigned based on the generated flash-mob events. The number of posts related to an event is correlated with the intensity

of the given event. The volume of activity around these events is modeled following a model similar to that described in [17]. Figure 5.8 shows the distribution of the activity around each flash-mob event. The x axis represents the number of days around the peak. The y axis represents the activity volume in that moment. We use this distribution to assign the creation date to flash-mob posts and comments.

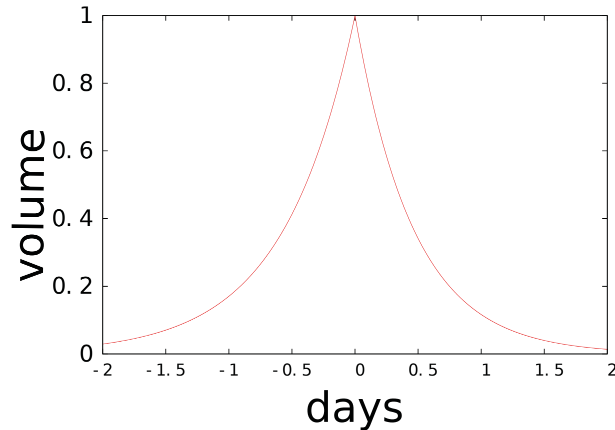


Figure 5.8: Flash-mob posts volume activity.

Furthermore, in order to reproduce the more uniform every day's user activity, DATAGEN also generates post uniformly distributed along all the simulated time.

5.4 Determinism

When designing a navigational benchmark, having a deterministic dataset generator is very important. As we have seen, different input values for the same query can translate to different execution plans of diverse complexity. To make the comparison of the different systems implementing the benchmark more fair, we need to ensure that they operate on the same dataset for a given scale factor, to eliminate the possibility that a system may execute easier queries than another.

DATAGEN is implemented to produce deterministic results, regardless the number of used machines and the platform used. DATAGEN achieves that by using two key techniques: the use of pseudo-random number generators, and by dividing the generator process into logical blocks.

First, using pseudo random number generators allows us to produce sequences of numbers with similar properties to those found in random numbers, but deterministically: two pseudo-random number generators fed with the same feed will produce the same sequence of numbers. Second, DATAGEN divides the generation process into virtual blocks of the same size with a unique block id. This block id is used to feed all the pseudo-random number generators used when processing that block, so regardless of when the block is generated, and the machine assigned to generate it, the generated data is the same.

5.5 Time-Correlated entity id generation

While how the ids of the entities are generated could be seen as a minor issue, in reality, it can have a great impact on the performance of the systems implementing the benchmark. In real systems, internal ids assigned to entities are typically correlated with the time at which the entity was created. One may think of a pool of unused unique ids, which are retrieved from whenever a new entity of a given type is created. These ids could be consecutive ids, or some sort of combination of data (e.g. concatenation of a unique id plus a creation date) that could ensure id uniqueness. Having ids created in such a way allows for two things: placing data in memory correlated with creation time and therefore, some queries could benefit from it as would improve data locality, and ease data load time as these would be inserted in consecutive data pages.

In DATAGEN we emulate this feature by playing with the bits of ids, to include the creation date. In DATAGEN there are three types of ids, those assigned to Persons, those assigned to Messages and those assigned to entities created offline (Organizations, Countries, Tags, etc ...).

Person ids are generated to encompass several information: the creation date, the person index and the degree percentile, this last one used for internal analysis purposes. Figure 5.10 shows how the bits of persons are created. Creation date bits (bits 48 to 41) are selected from dividing the simulated time frame into 256 buckets, and selecting the bucked where the person creation date falls. The person index is limited to 34 bits (bits 40 to 7). This limitation arises from how message ids are created as we will discuss shortly. Finally, the first seven bits (bits 6 to 0) are used to encode the friendship degree percentile the person is falling.

	date bucket 48-41	person index 40-7	percentile 6-0

Figure 5.9: Bit composition of Person Id.

Message ids encompass the creation date, the block id and the message index within the block. Figure **??** shows how the different bits are used. The first twenty bits, from bit 19 to 0, are used to store the twenty lowest bits of the message index within the block. The next twenty bits, from bit 39 to 20 are used to store the block index. The next sixteen bits, from bit 56 to 40 are used to store the next sixteen bits of the message index within the block, and finally the last eight bits are used to store the date bucket where the message creation date falls, similarly to the person id.

date bucket 63-56	message index 16 middle bits 55-40	block id 39-20	message index 20 lower bits 19-0

Figure 5.10: Bit composition of Message Id.

Clearly, the amount of bits we devote to user index, message index and block id affects the maximum number of entities we can create. These bits impose several limitations. The first one is that the maximum number of messages that can be created within a block is 2^{36}. Since there are twenty bits devoted to block id, at most we can generate $2^{20} \cdot 2^{36} = 2^{56}$. Similarly, the number of Persons is limited by the number of blocks and the size of the block. Currently, DATAGEN has a block size of 10000, which means that the maximum number of Persons DATAGEN can generate without having an overflow or overlap in the generation of ids is $2^{20} \cdot 10000 \simeq 2^{34}$, which is exactly the amount of bits we devoted to store the Person index in the Person id. Of course, we could implement an adaptive id bit distribution, which depending on the size of the generated network, increase the number of persons per block, and therefore reduce the maximum number of logical blocks, and therefore having more bits per Messages and Persons.

Finally, as the upper bits of ids contain the date bucket, those entities created at similar dates will clearly fall into the same bucket, and therefore will contain the same upper bits.

5.6 Update Streams

Some real use cases of navigational queries and graphs in general, have periodic updates on their data. Therefore, a navigational benchmark must allow the possibility to test systems with updates. The reason why update streams must be generated by the data generator, is because the characteristics of the data must be maintained regardless of the updates being performed. If they were performed in a random fashion, for instance, at some point in the future the non-uniformness of the data produced could be lost during the update process. Therefore, DATAGEN produces, along with the dataset, a set of update streams that maintain the structure of the data. These update streams are generated by splitting the generated dataset into two parts: the static part which will be bulk loaded into the database and the dynamic part. The static part is roughly the 90% of the total generated network, while the dynamic part is the remaining 10%.

Especial care must be taken when generating the update streams, so not to put data depending on update events into the static part of the network. For instance, all friendship relations between a Person falling into the dynamic part must also be into the dynamic part of the network, as friendship relation cannot exist without their corresponding endpoint Persons.

Finally, consecutive dependent events in the dynamic part must be separated by a predefined delta time δt. For instance, if a Person is created at time t, then any update event involving this person, such as creating a Post/Comment or adding a relation to a friend, cannot occur before $t + \delta t$. This is necessary to allow the execution driver provided by LDBC to execute in a distributed environment and scale, as described in Deliverable 2.2.3 [13].

6 Workload

6.1 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.

6.2 Lookup Query Descriptions

Notes:

- Some queries require returning the content of a post. As stated in the schema, posts have content or imageFile, but not both. An empty string in content represents the post not having content, therefore, it must have a non empty string in imageFile and the other way around.

6.2.1 Query1 - Friends with a certain name

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

Person.id	ID
Person.firstName	String

- **Results:**

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

- **Relevance:** CP1.3 CP2.1 CP5.3 CP7.1
  This query is a representative of a simple navigational query. It looks for paths of length one, two or three through the knows relation, starting from a given Person and ending at a Person with a given name. It is interesting for several aspects. First, it requires for a complex aggregation, that is, returning the concatenation of universities, companies, languages and email information of the person. Second, 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. Third, an important part of the query execution time is expected to be spent into translating internal ids to external ones. Finally, performance is highly sensitive to properly estimating the cardinalities in each transitive path, and paying attention not to explode already visited Persons.

6.2.2 Query2 - Recent posts and comments by your friends

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

- **Parameters:**

Person.id	ID
date	DateTime

- **Results:**

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

- **Relevance:** CP1.1 CP2.2 CP2.3 CP3.2
  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.

6.2.3 Query3 - Friends and friends of friends that have been to countries X and Y

- **Description:** Given a start Person, find Persons that are their friends and friends of friends (excluding start Person) that have made Posts/Comments in the given Countries X and Y within a given period. Only Persons that are foreign

to Countries X and Y are considered, that is Persons whose Location is not Country X or Country Y. Return top 20 Persons, and their Post/Comment counts, in the given countries and period. Sort results descending by total number of Posts/Comments, and then ascending by Person identifier.

- **Parameters:**

Person.id	ID	
CountryX.name	String	
CountryY.name	String	
startDate	Date	// beginning of requested period
duration	32-bit Integer	// duration of requested period, in days // the interval [startDate, startDate + Duration) is closed-open

- **Results:**

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

- **Relevance:** CP2.1 CP3.1 CP5.1
  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.

6.2.4 Query4 - New topics

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

Person.id	ID	
startDate	Date	
duration	32-bit Integer	// duration of requested period, in days // the interval [startDate, startDate + Duration) is closed-open

- **Results:**

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

- **Relevance:** CP2.3
  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.

6.2.5 Query5 - New groups

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

- **Parameters:**

Person.id	ID
date	Date

- **Results:**

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

- **Relevance:** CP2.3 CP3.3
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.

6.2.6 Query6 - Tag co-occurrence

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

- **Parameters:**

Person.id	ID
Tag.name	String

- **Results:**

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

- **Relevance:** CP5.1
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.

6.2.7 Query7 - Recent likes

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

- **Parameters:**

Person.id	64-bit Integer

- **Results:**

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

- **Relevance:** CP2.2 CP2.3 CP3.3 CP5.1 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.

6.2.8 Query8 - Recent replies

- **Description:** Given a start Person, find (most recent) Comments that are replies to Posts/Comments of the start Person. Only consider immediate (1-hop) replies, not the transitive (multi-hop) case. Return the top 20 reply Comments, and the Person that created each reply Comment. Sort results descending by creation date of reply Comment, and then ascending by identifier of reply Comment.
- **Parameters:**

Person.id	ID

- **Results:**

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

- **Relevance:** CP2.4 CP3.2 CP3.3 CP5.3
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.

6.2.9 Query9 - Recent posts and comments by friends or friends of friends

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

Person.id	ID
date	Date

- **Results:**

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

- **Relevance:** CP1.1 CP1.2 CP2.2 CP2.3 CP3.2 CP3.3
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.

6.2.10 Query10 - Friend recommendation

- **Description:** Given a start Person, find that Person's friends of friends (excluding start Person, and immediate friends), who were born on or after the 21st of a given month (in any year) and before the 22nd of the following month. Calculate the similarity between each of these Persons and start Person, where similarity for any Person is defined as follows:

  – common = number of Posts created by that Person, such that the Post has a Tag that start Person is Interested in
  – uncommon = number of Posts created by that Person, such that the Post has no Tag that start Person is Interested in
  – similarity = common - uncommon

  Return top 10 Persons, their Place, and their similarity score. Sort results descending by similarity score, and then ascending by Person identifier

- **Parameters:**

Person.id	ID	
month	32-bit Integer	// between 1-12

- **Results:**

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

- **Relevance:** CP2.3 CP3.3 CP4.1 CP4.2 CP5.1 CP5.2 CP6.1 CP7.1
  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.

6.2.11 Query11 - Job referral

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

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

- **Results:**

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

- **Relevance:** CP1.4 CP2.3 CP2.4 CP3.3
  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.

6.2.12 Query12 - Expert search

- **Description:** 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. Return top 20 Persons, the reply count, and the collection of Tags. Sort results descending by Comment count, and then ascending by Person identifier

- **Parameters:**

Person.id	ID
TagClass.id	ID

- **Results:**

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

- **Relevance:** CP1.5 CP3.3 CP7.2 CP7.3
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.

6.2.13 Query13 - Single shortest path

- **Description:** Given two Persons, find the shortest path between these two Persons in the subgraph induced by the Knows relationships. Return the length of this path.
  - -1 : no path found
  - 0: start person = end person
  - > 0: regular case
- **Parameters:**

Person.id	ID	// person 1
Person.id	ID	// person 2

- **Results:**

length	32-bit Integer

- **Relevance:** CP3.3 CP7.2 CP7.3
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.

6.2.14 Query14 - Weighted paths

- **Description:** Given two Persons, find all weighted paths of the shortest length between these two Persons in the subgraph induced by the Knows relationship. The nodes in the path are Persons. Weight of a path is sum of weights between every pair of consecutive Person nodes in the path. The weight for a pair of Persons is calculated such that every reply (by one of the Persons) to a Post (by the other Person) contributes 1.0, and every reply (by ones of the Persons) to a Comment (by the other Person) contributes 0.5. In the unlikely case that start and end date are the same Person, weight is 0. Return all the paths with shortest length, and their weights. Sort results descending by path weight. Ordering of paths of the same length is not specified.
- **Parameters:**

Person.id	ID	// person 1
Person.id	ID	// person 2

- **Results:**

[Person.id]	[ID]	// Identifiers representing an ordered sequence of the Persons in the path
weight	64-bit Float	

- **Relevance:** CP3.3 CP7.2 CP7.3
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.

6.3 Update Query Descriptions

6.3.1 Query1 - Add Person

- **Description:** Add a Person to the social network.
- **Parameters:**

Person.id	ID
Person.firstName	String
Person.lastName	String
Person.gender	String
Person.birthDay	Date
Person.creationDate	DateTime
Person.locationIp	String
Person.browserUsed	String
Person-isLocatedIn->City.id	ID
Person.speaks	{ String }
Person.emails	{ String }
Person-hasInterest->Tag.id	{ ID }
{ Person-studyAt->University.id, Person-studyAt->.classYear }	{ID, 32-bit Integer}
{ Person-workAt->Company.id, Person-workAt->.workFrom }	{ID, 32-bit Integer}

6.3.2 Query2 - Add Friendship

- **Description:** Add a friendship relation to the social network
- **Parameters:**

Person.id	ID	// person 1
Person.id	ID	// person 2
Person-knows->.creationDate	DateTime	

6.3.3 Query3 - Add Forum

- **Description:** Add a Forum to the social network.
- **Parameters:**

Forum.id	ID	// person 1
Forum.title	String	// person 2
Forum.creationDate	DateTime	
Forum-hasModerator->Person.id	{ ID }	
Forum-hasTag->Tag.id	{ ID }	

6.3.4 Query4 - Add Forum Membership

- **Description:** Add a Forum membership to the social network.
- **Parameters:**

Person.id	ID
Person-hasMember->Forum.id	ID
Person-hasMember->.joinDate	DateTime

6.3.5 Query5 - Add Post

- **Description:** Add a Post to the social network.

- **Parameters:**

Post.id	ID
Post.imageFile	String
Post.creationDate	DateTime
Post.locationIp	String
Post.browserUsed	String
Post.language	String
Post.content	Text
Post.length	32-bit Integer
Post-hasCreator->Person.id	ID
Forum-containerOf->Post.id	ID
Post-isLocatedIn->Country.id	ID
{Post-hasTag->Tag.id}	{ID}

6.3.6 Query6 - Add Like Post

- **Description:** Add a Like to a Post of the social network.
- **Parameters:**

Person.id	ID
Post.id	ID
Person-likes->.creationDate	DateTime

6.3.7 Query7 - Add Comment

- **Description:** Add a Comment replying to a Post/Comment to the social network.
- **Parameters:**

Comment.id	ID	
Comment.creationDate	DateTime	
Comment.locationIp	String	
Comment.browserUsed	String	
Comment.content	Text	
Comment.length	32-bit Integer	
Comment-hasCreator->Person.id	ID	
Comment-isLocatedIn->Country.id	ID	
Comment-replyOf->Post.id	ID	// -1 if the comment is a reply of a comment.
Comment-replyOf->Comment.id	ID	// -1 if the comment is a reply of a post.
{Comment-hasTag->Tag.id}	{ID}	

6.3.8 Query8 - Add Like Comment

- **Description: Add a Like to a Comment of the social network.**
- **Parameters:**

Person.id	ID
Comment.id	ID
Person-likes->.creationDate	DateTime

6.4 Choke-point coverage

Table 6.1 shows a summary of the different choke-points covered by each query. The two principal observations are drawn from the table: all proposed choke-points are covered by at least one query, and that choke-point distribution is not uniform, with CP2.3, CP3.3, CP5.1, CP7.2 and CP7.3 at the top of the coverage ranking. This result is not casual. If we look in detail what these choke-points stress, we see they are very influenced by the nature of navigational queries. For instance, CP3.3 tests the ability of the execution engine for efficient indexes that can work well when accesses are scattered. This clearly is affected by the complexly structured nature of the data used by navigational queries and the patterns they are looking for. In the case of CP2.3, it stresses the ability of the query optimizer to choose the appropriate join type to perform, which will depend on

the cardinality of intermediate results, which are heavily influenced by the skewed distributions of graph data (which where deliberately reproduced by DATAGEN). CP5.1 is related to correlated sub-queries, which are common in many types of applications and CP7.2 and CP7.3 are related to computing paths, which are typical patterns searched in navigational query applications. This table confirms that the proposed queries are valid to test the most important aspects of navigational query based systems, reproducing the challenges these queries pose.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14
CP1.1		*							*					
CP1.2									*					
CP1.3	*													
CP1.4								*			*			
CP1.5												*		
CP2.1	*		*											
CP2.2		*					*		*					
CP2.3		*		*	*		*		*	*	*			
CP2.4								*			*			
CP3.1			*											
CP3.2		*						*	*					
CP3.3					*		*	*	*	*	*	*	*	*
CP4.1										*				
CP4.2										*				
CP5.1			*			*	*			*				
CP5.2										*				
CP5.3	*													
CP6.1										*				
CP7.1										*				
CP7.2												*	*	*
CP7.3	*											*	*	*

Table 6.1: Choke-point coverage

6.5 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 behavior 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 sub-queries 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 run times 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. For a more detailed description on how substitution parameter curation works, please read Deliverable 2.2.4 [14]:

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 run times. 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.

6.6 Load Definition

LDBC-SNB Test Driver is in charge of the execution of the Interactive Workload. At the begining 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 choosen 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. 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 assigned a different interleave time. This interleave time corresponds to the amount of time that must elapse between issuing two query instances of the same type. Interleave times have been empirically determined by experimenting with the workload on different existing database technologies. Those more complex query types, will have larger interleave times than those faster queries. Table 6.2 shows the current interleave times assigned to each query type in milliseconds.

Query Type	ms	Query Type	ms
Query 1	30	Query 8	1
Query 2	12	Query 9	40
Query 3	72	Query 10	27
Query 4	27	Query 11	18
Query 5	42	Query 12	34
Query 6	18	Query 13	1
Query 7	13	Query 14	66

Table 6.2: Interleaved latencies for each query type in miliseconds.

The specified interleave times, 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" further apart the queries of the workload. This is achieved by means of a factor that is multiplied by the interleaved latencies. Therefore, different throughput can be tested while maintaining the relative ratios between the query types. For a more detailed information about the driver and how the workload is generated, please see Deliverable 2.2.3 [13].

7 Performance Metrics

When measuring system performance the objective is to gain insight into how that system would perform in a known, realistic scenario. Data obtained from such testing is useful for capacity planning, setting of service level agreements (SLA), software selection, etc. In all cases the desired information is: how quickly (latency) will the system under test (SUT) respond to a request, when subjected to that scenario. Knowing the expected latency for an SUT under some given load is usually more valuable than knowing the maximum throughput of that same SUT.

As such, and especially given the interactive nature of the SNB workload, the LDBC-SNB Test Driver priotizes the measurement of latency, over throughput; it controls the rate at which queries are issued to the SUT, then measures and records the latency of each query executed during that workload.

A side note, although controlling the rate of query executions allows LDBC-SNB Test Driver to more accurately monitor latencies, and to generate precise, repeatable workloads (a necessity when systems are to be directly compared to one another), various measurement-related challenges remain. Measuring latencies in a correct and statistically meaningful way is difficult, the LDBC-SNB Test Driver has been designed in such a way as to avoid measurement-related errors due to, for example, Coordinated Omission. For more on these challenges and the way they are overcome refer to Deliverable 2.2.3 [13].

7.0.1 Recorded Metrics

Various metrics can be used to report latencies, starting with mean, median, min, and max. These are each useful, but for populations of data can be misleading.

Mean and median are useful for seeing what a 'normal' measurement may look like, but they tend to hide outliers. Results reported as mean values are typically smoothened, where peaks and troughs are less extreme, or hidden entirely. On the contrary, min and max are useful for communicating the most extreme cases, but due to this they are very sensitive, affected by a single outlier.

Standard deviation is another alternative, however it is frequently misinterpreted. For example, a common assumption is that the interval of mean +/- three standard deviations contains 99.73% of the measurements. This is a misuse of the three-sigma rule, in most cases resulting in an 'optimistic' result. In reality it is often the case that fewer than 99.73% of measurements are within this interval, the higher percentiles are worse than the calculation claims, meaning that the system is performing worse than thought.

The three-sigma/empirical rule states that 99.73% of values lie within three standard deviations of the mean, but *only for normal distributions*. Latency profiles of complex systems rarely exhibit normal distributions, more commonly they are some variant of a multi-modal distribution (continuous probability distribution with two or more modes), e.g., a system where latencies can be roughly grouped into three distinct categories (three modes): low (cached data), medium (regular case), high (corner cases the system is not optimized for).

Regardless, it is safer to record metrics in such a way that is distribution-independent. Percentiles provide such an alternative. Percentiles communicate the value below which a certain percentage of the data is included. For example, a 95th percentile is the value which is greater or equal to 95% of the data. Irrespective of distribution, percentiles are useful for exposing some percentage of outlier measurements, while negating the extreme sensitivity of max (e.g., 100th percentile) measurements.

Given the advantages and disadvantages of these different approaches, LDBC-SNB Test Driver reports benchmark results using a combination of metrics, making it possible to perform analysis along different dimensions, therefore providing a broader view of system performance.
The following aggregate metrics are reported:

- **Workload start time**: wall clock time when when execution of the first query began
- **Workload finish time**: wall clock time when execution of the last query completed
- **Total workload duration**: duration between workload start time and workload finish time
- **Total query count**: number of query executions, including queries of all types

In addition, the following detailed, per query type metrics are also reported:

- **Query count**: number of query executions, by query type
- **Min latency**: minimum recorded latency, by query type
- **Max latency**: maximum recorded latency, by query type
- **50th percentile latency**: 50th percentile recorded latency, by query type
- **90th percentile latency**: 50th percentile recorded latency, by query type
- **95th percentile latency**: 50th percentile recorded latency, by query type
- **99th percentile latency**: 50th percentile recorded latency, by query type

References

[1] Renzo Angles, Arnau Prat-Pérez, David Dominguez-Sal, and Josep-Lluis Larriba-Pey. Benchmarking database systems for social network applications. In *First International Workshop on Graph Data Management Experiences and Systems*, page 15. ACM, 2013.

[2] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 international conference on Management of data*, pages 1185–1196. ACM, 2013.

[3] Sotirios Beis, Symeon Papadopoulos, and Yiannis Kompatsiaris. Benchmarking graph databases on the problem of community detection. In *New Trends in Database and Information Systems II*, pages 3–14. Springer, 2015.

[4] Peter Boncz, Thomas Neumann, and Orri Erling. Tpc-h analyzed: Hidden messages and lessons learned from an influential benchmark. In *Performance Characterization and Benchmarking*, pages 61–76. Springer, 2014.

[5] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, volume 4, pages 442–446. SIAM, 2004.

[6] Marek Ciglan, Alex Averbuch, and Ladialav Hluchy. Benchmarking traversal operations over graph databases. In *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pages 186–189. IEEE, 2012.

[7] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)*, 25(2):73–169, 1993.

[8] Gremlin. Gremlin - graph traversal language. http://gremlin.tinkerpop.com, 2014.

[9] Steve Harris and Andy Seaborne. Sparql 1.1 query language. *W3C working draft*, 14, 2010.

[10] Huahai He and Ambuj K Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 405–418. ACM, 2008.

[11] InfiniteGraph. Infinitegraph database. http://objectivity.com/infinitegraph, 2014.

[12] LDBC. Ldbc deliverable 2.2.2: Data generator. http://www.ldbc.eu:8090/download/attachments/1671227/d2.2.2-final.pdf, 2014.

[13] LDBC. Ldbc deliverable 2.2.3: Benchmarking transactions, 2014.

[14] LDBC. Ldbc deliverable 2.2.4: Benchmarking complex queries, 2014.

[15] LDBC. Ldbc deliverable 3.3.2: Graph database infrastructure and language expressivity. http://www.ldbc.eu:8090/download/attachments/1671227/d3.3.2.pdf, 2014.

[16] LDBC. Ldbc-snb specification. https://github.com/ldbc/ldbc_snb_docs, 2014.

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

[18] Peter Macko, Daniel Margo, and Margo Seltzer. Performance introspection of graph databases. In *Proceedings of the 6th International Systems and Storage Conference*, page 18. ACM, 2013.

[19] Guido Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. 2008.

[20] Neo4j. Neo4j ldbc-snb interactive workload implementation. https://github.com/ldbc-dev/ldbc_snb_workload_interactive_neo4j, 2014.

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

[22] Nataliia Pobiedina, Stefan Rümmele, Sebastian Skritek, and Hannes Werthner. Benchmarking database systems for graph pattern matching. In *Database and Expert Systems Applications*, pages 226–241. Springer, 2014.

[23] Arnau Prat-Pérez, David Dominguez-Sal, and Josep L Larriba-Pey. Social based layouts for the increase of locality in graph operations. In *Database Systems for Advanced Applications*, pages 558–569. Springer, 2011.

[24] Sherif Sakr, Sameh Elnikety, and Yuxiong He. G-sparql: a hybrid engine for querying large attributed graphs. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 335–344. ACM, 2012.

[25] Mauro San Martın, Claudio Gutierrez, and Peter T Wood. Snql: A social networks query and transformation language. *cities*, 5:r5, 2011.

[26] Neo Technologies. Neo4j cypher documentation. http://docs.neo4j.org/chunked/milestone/cypher-query-lang.html, 2014.

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

[28] Julian R Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.