Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

LDBC Social Network Benchmark Interactive v2.0

Author: David A. Püroja (10469036)

1st supervisor: prof. dr. Peter A. Boncz (CWI, VU)
daily supervisor: dr. Gábor Szárnyas (CWI)
2nd reader: prof. dr. Hannes F. Mühleisen (CWI, RU)

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computational Science

January 23, 2023

Qui Patitur Vincit

Abstract

The Linked Data Benchmark Council (LDBC) maintains the Social Network Benchmark Interactive (SNB-I) workload, which targets transactional database management systems capable of supporting graph workloads. The first version of SNB-I, released in 2015, has implementations in a number of systems, including the Neo4j and TuGraph graph databases, as well as the PostgreSQL and Umbra relational database management systems. While this benchmark was influential, it had a number of shortcomings, including the lack of delete operations and query variants, as well as the limited scalability of its data and parameter generators.

In this thesis, we present the design of the SNB-I v2.0 workload. This renewed workload includes data sets up to 30× larger, includes deep deletion operations, and supports generating parameters in temporal buckets to ensure predictable runtimes. Additionally, we present a design for a distributed driver that allows it to stress scale-out systems in the future.

We used a graph DBMS and two relational DBMSs to conduct experiments with the new workload to study its characteristics. First, we performed a three-way cross-validation to ensure the correctness of the implementations. Second, we evaluated the performance impact of the delete operations and found that they effect graph systems less significantly than relational ones. Finally, we investigated the scalability of the new temporal parameter generation and the stability of the resulting query execution times. The experiments show that the new parameter generation scales well when extending to larger scale factors, achieving up to 100× better performance compared to v1.0. We found that the new parameter generation improves the stability of runtimes for chosen parameters.

The resulting workload is available as open-source software with unit tests, a continuous integration pipeline, and contributor guidelines. Additionally, we improved usability of the benchmark with providing deployment and provisioning scripts using Terraform and Ansible to setup benchmark environments in the cloud.

Acknowledgements

First, I would like to thank my daily advisor, Gábor Szárnyas, for all the suggestions, insights, *gezelligheid*, and ideas when I got stuck during the thesis. I am grateful for your patience in answering my questions and the late-night debugging sessions together. Without your help, it would have been impossible to complete this thesis. Peter Boncz, thank you for the opportunity to do my thesis at CWI and for traveling to Philadelphia to attend SIGMOD'22 and also CIDR'23 in Amsterdam, which certainly broadened my horizons in the field of data systems.

In addition to my supervisors, I would like to thank Arvind Shyamsundar from Microsoft, and Altan Birler from TUM for advising on the implementations for SQL Server and Umbra, respectively.

I want to thank my family for all the support they gave me during the period of the thesis. Finally, I would like to thank my girlfriend, Kayleigh, for all her support, "work-free" weekend trips, dinners and being there for me.

Part of the Azure cloud infrastructure used for the experiments in this thesis is funded by the Scalable Graph Workload grant from SURF.

Contents

List of Figures

LIST OF FIGURES

LIST OF FIGURES

List of Tables

LIST OF TABLES

1

Introduction

1.1 Context

Graph data processing is becoming more prevalent in many application domains where highly-connected data sets are present, for example, in finance, infrastructure, communication networks, and social networks (2, 53, 54). In light of this, there is an emergence of graph database management systems (GDBMSs), e.g., Neo4j (40), Nebula Graph (73), Amazon Neptune (7), and Kùzu (19). These systems use query languages created for graph data processing to enable queries such as traversal and shortest paths. Examples of these query languages are Cypher (23) and the upcoming ISO GQL (Graph Query Language) standard (16).

To help speed up the adoption of graph(-capable) data management systems, a group of vendors and researchers founded the Linked Data Benchmark Council (LDBC) (4). LDBC aims to stimulate technological progress among competing systems. To this end, they specify benchmarks capturing graph workloads, define the benchmarking procedure, and verify benchmark results via audits (18). The benchmarks set a challenge and develop technology around databases processing graph data, accelerating these systems' maturity and adding graph features to existing systems. The LDBC maintains the Social Network Benchmark (SNB) suite, containing graph workloads to benchmark database systems. This benchmark suite consists of the Interactive workload (SNB-I) and Business Intelligence workload (SNB-BI) (61). SNB-I focuses on transactional graph processing: it executes read queries while continuously inserting new data into the graph. The SNB-BI focuses on aggregation- and join-heavy complex queries touching a large portion of the graph and includes microbatches of insert and delete operations (1). While both workloads have updates, updates in SNB-I

are executed concurrently while SNB-BI allows batched execution; concurrent updates are optional. The SNB workloads make use of parameterized query templates: the benchmark use parameters with similar runtime behavior to execute queries, which are selected using parameter curation (28).

The LDBC SNB benchmarks use a choke-point-based benchmark design process (18). These choke points are well-chosen difficulties introduced by the workload, identified as crucial technical challenges. These choke points cover complex problems which are not always naturally part of synthetic benchmarks and incentivize database developers to employ optimization techniques in the database to increase performance (10). An example of a choke point is the ability of the DBMSs to perform query optimizations like pushdown path restrictions on an existing path index or to compute the unweighted shortest paths between a node and a set of nodes.

While there are other benchmarks targeting graph database systems like LinkBench (5) and GDB-test (35), the SNB Interactive is unique since it includes concurrent updates and inserting data with inter-dependencies, which requires a complex driver able to track these dependencies. In addition, the benchmark includes highly-connected data with realistic attributes and structure correlations, and it contains graph queries like hierarchy traversal, multi-hop neighborhood queries, and shortest paths.

Emerging graphs are also frequently found in tabular data sets (53). To analyze these in-place, graph extensions were proposed to relational database management systems (RDBMSs), e.g., SAP HANA Graph (52), IBM Db2 Graph (63), Microsoft SQL Server[1], and DuckDB (62). Several of these systems use an extension to their existing SQL dialect, e.g., the Graph Extension is integrated into Microsoft SQL Server's T-SQL language. Additionally, the upcoming ISO SQL:2023 standard introduces the SQL/PGQ (Property Graph Queries) (16) extension which shares its graph query syntax with GQL. These languages originate from a liaison between LDBC and the International Organization for Standardization (ISO), where LDBC makes technical contributions and is actively involved in the standard.

We focus in this research on the Interactive workload. The current version of SNB Interactive, v1.0, has the following limitations. First, it does not support deletions. Therefore,

[1] `https://docs.microsoft.com/en-us/sql/relational-databases/graphs/sql-graph-overview`

it does not test the performance of dynamic graph processing systems under such operations (71). Deletions in GDBMSs are also desirable with the General Data Protection Regulation (GDPR) (56), which gives citizens of the EU the right to have their personal information removed from personal-data processing database systems. Second, the scalability of Interactive v1.0 was limited to graphs of 1 TB. In this thesis, we introduce support for larger data sets up to 30 TB to make the benchmark capable of stressing the scalability of database systems. The Interactive driver runs the queries in parallel against the DBMS, which mostly support concurrent transactions, to simulate realistic transactional workloads. Since the execution of the queries and updates are concurrent, the results of queries are not deterministic. Therefore, for the update queries, the benchmark makes use of dependency tracking: each update operation has a timestamp denoting the creation time of the operation it depends on. Not including dependency tracking while issuing queries concurrently can lead to errors, e.g., trying to insert a reply to a post that is not present in the database.

This research has several challenges. First, we must refactor the driver to include delete queries, support for larger scale factors, and design an update stream generator with dependency tracking capable of handling deletes. Implementing deletions in the benchmark driver is non-trivial since operations on the database may depend on each other. Figure 1.1 gives an example of a deletion in the graph. To prevent the execution of a query dependent on a deleted or not yet inserted entity or edge, all benchmark driver processes issuing queries regarding that specific entity must know whether they can execute a query. Dependency tracking is challenging to implement in a scalable environment since the communication of the dependency time to all processes is expensive, thus a potential bottleneck for the driver.

The parameters selected for the complex queries require temporal attributes to know when a parameter can execute to ensure runtimes are predictable. Temporal attributes require a parameter generator that can select parameters with the desired properties for a given time during the simulation time of the benchmark. Additionally, we must include support for temporal parameters in the driver. The changes in the driver necessitate updating the reference implementations to support the new delete queries. For the distributed driver design, partitioning of the operations needs to be taken into account. The operations should be partitioned so each client has approximately the same amount of workload.

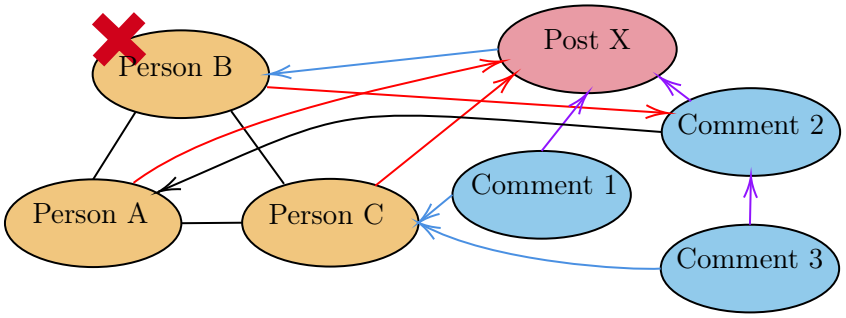

Figure 1.1: Example of a deletion in the SNB graph. The red edges are likes from a person to a post/comment, blue edges are creator edges, purple edges are replyOf edges and black edges are knows edges. In case Person B is removed from the graph, the cascading effects cause all nodes and edges to be deleted except Person A, Person C and their knows edge, yielding a deletion of 5 nodes and 12 edges.

The result is the SNB Interactive v2.0 which adds support for deep deletions, temporal parameters, and scalability for large data sets, making it relevant for more powerful systems in the future. Additionally, we present a distributed driver architecture.

1.2 Research Questions

This thesis focuses on designing and implementing the LDBC SNB-I v2.0 to include deletions. As the basis for the SNB-I v2.0, we use the driver and implementations from v1.0 including deletions used in the SNB-BI workload.

1. How can deletions be integrated into the Social Network Benchmark Interactive?
   (a) What changes in the Interactive benchmark need to be implemented to support deletions?
   (b) What changes are required in the Interactive benchmark driver to support deletions?
   (c) Do the data sets produced by the Spark Datagen preserve the characteristics of the ones generated by the Hadoop Datagen?
2. How to generate parameters with similar behavior to the query template with the inclusion of inserted and deleted nodes and edges?
   (a) What effect do deleted entities have on the parameter generation?
   (b) How to implement temporal parameter generation?

3. What effect does the inclusion of deletion operations have on the performance of the systems under test?
4. How can the driver be made scalable?
   (a) How to scale up the Interactive benchmark driver?
   (b) How can the Interactive benchmark driver be made distributed?

1.3 Thesis Structure

The thesis is structured as follows. In Chapter 2, we provide background information on the current LDBC SNB Benchmark, giving an overview of the Datagen, driver, and implementations currently used in SNB-I v1.0. Chapter 3 gives the design and implementation details, describing the changes made in the driver to support the new Datagen, the implementation of deletes, and the temporal parameter selection. In Chapter 4 the results of experiments are discussed, showing the effect of deletes on selected systems and a discussion on the parameters used to generate and curate parameters. Chapter 5, shows related work in the field of benchmarks containing graph workloads. In Chapters 6–9, we elaborate on the design and considerations for a distributed driver. Finally, in Chapter 10 and Chapter 11, the conclusions and answers to the research questions are given, along with several suggestions for future work.

2

Background

This section provides the context and concepts required for understanding the landscape of database benchmarks and graph data management, the related work, the current working of the LDBC SNB Interactive, and the design and implementation choices made in this thesis. First, Section 2.1 outlines the challenges of benchmarking database management systems (DBMSs). Second, Section 2.2 is a generic overview of graph data models. Then, in Section 2.3 we discuss different graph query languages and in Section 2.4 we present DBMSs supporting graph workloads. We give a definition of scalability in Section 2.5. Finally, we describe the Social Network Benchmark with its components in Section 2.6.

2.1 Benchmarking Database Management Systems

Performance metrics derived from benchmark results are used frequently in scientific papers and by vendors to compare their systems to other solutions. However, fair performance benchmarking is not trivial, and providing reproducible and interpretable results is often lacking (30). Raasveldt et al. (48) describe common pitfalls of benchmarking different database management systems, including non-reproducibility, failure to optimize, overly-specific tuning, and differentiation between cold, warm and hot runs.

No single metric can measure computer systems' performance on all applications: a system's performance may vary depending on the application domain. Therefore, domain-specific, application-level benchmarks are needed to gain insight into the system's expected performance on a given workload (55). Jim Gray defined the following four criteria in the Benchmark Handbook (26): *Relevance*, the benchmark must measure the peak performance and price/performance of systems when performing operations typical for a given

workload. *Portability*, the benchmark should be implementable on different systems and architectures. *Scalability*, the benchmark should apply to small and large computer systems. *Simplicity*, the benchmark should be understandable and implementable in a reasonable amount of time.

Database benchmarks not only allow testing of different technologies but also stimulate technological advances and provide a common understanding of important challenges that the database systems community should work on (33, 43). To guide the design of the benchmarks, "choke points" are used. *Choke points* are the technological challenges underlying a benchmark whose resolution will significantly improve the performance of a product across multiple workloads. In addition, choke points in benchmarks should point in relevant directions where technological advances are needed (10). The LDBC Social Network Benchmark suite contains several choke points, which are explained in Section 2.6.2.

2.2 Graph Data Models

In this section, we give an overview of popular graph query languages and illustrate them using the example graph query shown in Figure 2.2. This query retrieves the forum and moderator person for a given message.

2.2.1 Labeled Property Graph Model

The property graph model represents a graph by a set of nodes with relationships (edges), properties and labels. (50) The nodes contain properties, for example, a person has the properties email address, birthday, languages a person speaks. Relationships connect two nodes, e.g., a friendship. A relationship can also have properties, like the date a friendship connection is formed. Labels can be added to nodes and relationships. An example of a labeled property graph is given in Figure 2.1. Systems that support this model are, e.g., Neo4j (40) and TuGraph (67).

2.2.2 Resource Description Framework (RDF)

The *Resource Description Framework* (RDF) is used to express information about resources (entities). The data model consists of *triples:* a subject, a predicate and an object (13). A subject and object are the nodes in the graph, the predicates are the edges. The components can be globally described using an *Internationalized Resource Identifier* (IRI). In case of the subject or object, they can also have no IRI, which are referred to as *blank*

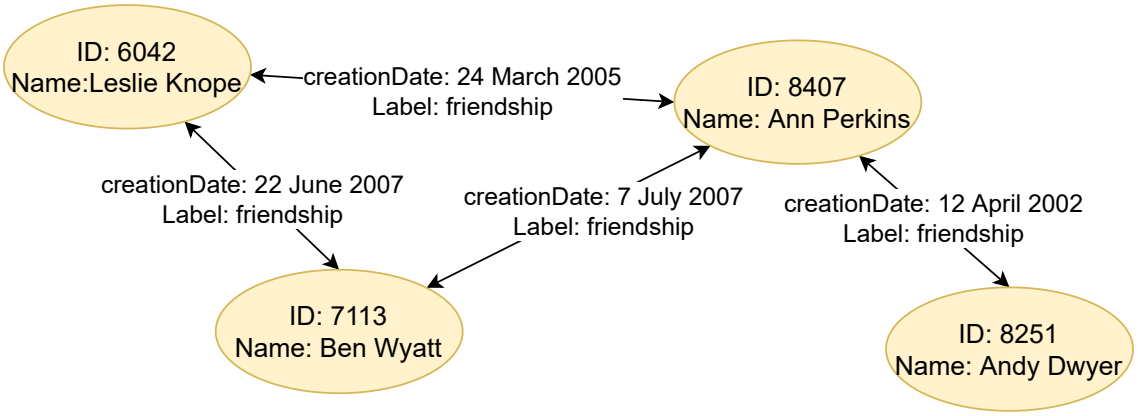

Figure 2.1: Example of a property graph. Yellow ellipses show the Person nodes with the properties ID and Name, connected by relationships with the creation date of the friendship as property.

nodes. Lastly, an object can also be a literal, which is used to express a value, such as strings, numbers and dates (13). Applications of RDF are for example found in data integration and federation.

2.3 Graph Query Languages

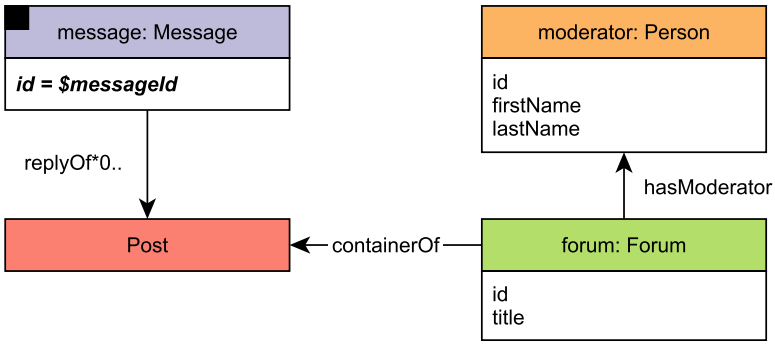

Figure 2.2: Example query: LDBC SNB Interactive short query 6 (1). The input parameter is denoted as `$messageId`.

2.3.1 Cypher

Cypher (24) is a query language, originally developed by Neo4j (72), used to query database systems with the property graph model. The language allows for "ASCII art" style of expressing a query to describe the graph pattern matching mechanism (24), also known as

visual graph syntax. An example is shown in Listing 2.1. It is now used by several commercial database systems besides Neo4j, e.g., AWS Neptune (7), Memgraph (37), TuGraph (67) and SAP HANA (52).

```
MATCH (message:Message {id: $messageId})-[:REPLY_OF*0..]->(rootPost:Post)
        <-[:CONTAINER_OF]-(forum:Forum)-[:HAS_MODERATOR]->(moderator:Person)
RETURN
    forum.id AS forumId,
    forum.title AS forumTitle,
    moderator.id AS moderatorId,
    moderator.firstName AS moderatorFirstName,
    moderator.lastName AS moderatorLastName
```

Listing 2.1: Example of query written in Cypher to retrieve information about a forum a message is posted in (LDBC Short query 6)

2.3.2 SQL/PGQ

SQL/PGQ (Property Graph Queries) (16) is an extension in the SQL:2023 standard enabling graph pattern matching and path-finding in SQL. Conceptually, SQL/PGQ represents the graph as vertex tables and edge tables, defined over the relational tables. It is a read-only extension, meaning that SQL/PGQ does not contain syntax to create new graphs, only how define graph views on SQL tabular schemas (16). An example is shown in Listing 2.2.[1]

```
SELECT forumId, forumTitle, moderatorId, moderatorFirstName, moderatorLastName
FROM GRAPH_TABLE (socialNetwork,
    MATCH (message:Message WHERE id = $messageId)-[:replyOf]->*(rootPost:Message)
            <-[:containerOf]-(forum:Forum)-[:hasModerator]->(moderator:Person)
    COLUMNS (forum.id AS forumId, forum.title AS forumTitle,
    moderator.id AS moderatorId, moderator.firstName AS moderatorFirstName,
    moderator.lastName AS moderatorLastName)
) gt;
```

Listing 2.2: Example query written in SQL/PGQ to retrieve information about a forum a message is posted in (LDBC Short query 6)

2.3.3 GQL

GQL, *Graph Query Language* (16), is a standalone property graph query language complementing SQL/PGQ, sharing the graph data model and graph pattern matching sublanguage with SQL/PGQ. In addition to SQL/PGQ, which only specifies a graph view on a table and outputs a table, GQL can also output graph views and create new graphs. The standard is still in development and is expected to release in March 2024.

[1] Note that in comparison with Cypher, the Kleene star, `*`, denotes a path of length `0..*` while in Cypher it denotes `1..*`.

2.3.4 Gremlin

Gremlin is a graph traversal language from the Apache TinkerPop project (22, 51). It contains three components: a graph, a traversal and a set of traversers. The latter move around the graph given the instructions in the traversal until all traversers have halted. The location of the halted traversers then gives the result (51). The language can be implemented in multiple host languages such as Java and Python, and it supports imperative and declarative styles. An example is shown in Listing 2.3[1].

```
g.V().has('Post','id',:messageId).fold()
 .coalesce(unfold(),V().has('Comment','id',:messageId)
 .repeat(out('replyOf').simplePath()).until(hasLabel('Post')))
 .in('containerOf').as('forum').out('hasModerator').as('moderator')
 .select('forum','moderator').by(valueMap('id','title'))
 .by(valueMap('id','firstName','lastName'))
```

Listing 2.3: Example of query written in Gremlin to retrieve information about a forum a message is posted in (LDBC Short query 6)

2.3.5 SPARQL

SPARQL is a query language to query and update graph data in RDF sources (69). A SPARQL query can access multiple data sources using a federated query, allowing merging data from multiple sources (69). Example of systems that support SPARQL is AWS Neptune (7) and Ontotext GraphDB (42). An example is shown in Listing 2.4[2].

```
PREFIX rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:  <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:   <http://www.w3.org/2001/XMLSchema#>
PREFIX sn:    <http://www.ldbc.eu/ldbc_socialnet/1.0/data/>
PREFIX snvoc: <http://www.ldbc.eu/ldbc_socialnet/1.0/vocabulary/>

SELECT
    ?forumId ?forumTitle ?moderatorId ?moderatorFirstName ?moderatorLastName
WHERE {
    BIND( $messageId AS ?messageId ) ?message snvoc:id ?messageId .
    OPTIONAL {
        ?message snvoc:replyOf* ?originalPostInner .
        ?originalPostInner a snvoc:Post .
    } .
    BIND( COALESCE(?originalPostInner, ?message) AS ?originalPost ) .
    ?forum snvoc:containerOf ?originalPost .
    ?forum snvoc:id ?forumId .
    ?forum snvoc:title ?forumTitle .
```

[1]Example from https://github.com/jackwaudby/ldbc-snb-implementations/blob/5e2992d6fe0cc2ea404c90b0405c6e6791da2049/gremlin_queries/short_read_6.txt

[2]Example from https://github.com/ldbc/ldbc_snb_interactive_impls/blob/c0db26133eb0b3837ee1b809975a5f797bc80dde/sparql/queries/interactive-short-6.sparql

```
19      ?forum snvoc:hasModerator ?moderator .
20      ?moderator snvoc:id ?moderatorId .
21      ?moderator snvoc:firstName ?moderatorFirstName .
22      ?moderator snvoc:lastName ?moderatorLastName .
23  }
```

Listing 2.4: Example of a query written in SPARQL to retrieve information about a forum a message is posted in (LDBC Short query 6)

2.4 DBMSs Supporting Graph Workloads

In this section, we discuss the key features of database management systems that have implementations for the LDBC SNB Interactive workload[1].

name	data model	query language	v1.0	v2.0
Neo4j	property graph	Cypher & API (Java)	yes	yes
Microsoft SQL Server	relational	T-SQL	yes	yes
PostgreSQL	relational	SQL	yes	yes
TuGraph	property graph	Cypher & API (C++)	yes	WIP
Umbra	relational	SQL	yes	yes

Table 2.1: List of selected database systems with an implementation of LDBC SNB Interactive. v1.0, v2.0: availability of implementations for LDBC SNB Interactive v1.0 and v2.0; WIP: work-in-progress.

2.4.1 TuGraph

TuGraph (67) is an open-source[2] GDBMS offered by the Ant Group, supporting property graph data structures. Cypher is a supported query language, as well as C++14 or Python stored procedures which are preloaded into the database. TuGraph has to date two audited LDBC SNB Interactive benchmark results.[3]

2.4.2 Neo4j

Neo4j is a GDBMS implemented in Java, which uses a "native" (pointer-based) graph data model to store the data. The data is represented as a node, edge, or attribute where nodes and edges can be labeled. (72) Neo4j provides three ways of querying graphs: through their graph query language Cypher, using the REST API, and programming against their Java

[1] LDBC SNB Implementations can be found at `https://github.com/ldbc/ldbc_snb_interactive_impls`

[2] `https://github.com/TuGraph-db/tugraph-db`

[3] `https://ldbcouncil.org/benchmarks/snb/`

API[1]. Neo4j has two editions: Community and Enterprise. Compared to the Community version, the Enterprise version comes with additional runtimes which support pipelined and parallel execution.

2.4.3 Umbra

Umbra is a flash-based RDBMS in development at the Technische Universität München (TUM), which uses columnar storage, and just-in-time compiled query execution where the logical query plan is compiled parallelized to machine code (41). It is a hybrid transaction-analytical processing system (HTAP) that can handle both OLAP and OLTP workloads.

2.4.4 PostgreSQL

PostgreSQL (27) is an open-source RDBMS using the SQL query language and capable of querying JSON-format representing nested (non-first normal form) data. Originally, it started as a project based on database Ingres at Berkeley (58). It supports add-ons, for example, PostGIS, adding support for geographic objects in SQL (47). It is a popular database with offers on major public cloud platforms as Azure, AWS, Google Cloud and IBM Cloud. To express path queries in PostgreSQL, the `SQL:1999 WITH RECURSIVE` statement is used.

2.4.5 Microsoft SQL Server

SQL Server is an RDBMS developed by Microsoft and uses a dialect of SQL, Transact-SQL, for querying data. Several editions are available depending on the features required by the user. Microsoft also offers SQL Server as a platform-as-a-service option through their Azure cloud platform as Azure SQL Database. However, the two offerings are only partially compatible: some query syntax is not fully interchangeable between standalone SQL Server and Azure SQL Database[2]. SQL Server supports HTAP workloads, providing OLTP and OLAP processing[3]. SQL Server has a graph extension, SQL Graph, which uses node and edge tables to store graph data models and uses the `MATCH` function to support pattern matching queries. In addition, SQL Graph includes the shortest path function.[4]

[1] `https://neo4j.com/developer/language-guides/`

[2] `https://learn.microsoft.com/en-us/azure/azure-sql/database/transact-sql-tsql-differences-sql-server?view=azuresql`

[3] `https://learn.microsoft.com/en-us/azure/architecture/data-guide/relational-data/online-analytical-processing`

[4] `https://learn.microsoft.com/en-us/sql/relational-databases/graphs/sql-graph-overview?view=sql-server-ver16`

2.5 Scalability

A benchmark should scale from small, single-node systems to large, distributed systems (26). There are two options when scaling systems, scaling up or scaling out. Scaling up is to optimize the application to maximize performance using one single-node system, for example, improving the application or upgrading the hardware of the single node. Scaling out is to make the application distributed across multiple systems. When deciding between these two scaling options, the COST-metric should be taken into account: Configuration that Outperforms a Single Thread (36), indicating how many machines one needs to achieve a performance equal to a single thread. The advantages of distributed systems are fault-tolerance and scaling out when required.

2.6 The LDBC Social Network Benchmark

The LDBC Social Network Benchmark (1) is a benchmark suite for assessing the performance of DBMSs with an emphasis on graph queries. The graph in SNB models a social network with properties similar to Facebook. The motivation to choose a social network as a benchmark scenario is because it is a graph-centric use case that is understandable and realistic as necessary for the benchmark queries. This social network introduces three non-trivial properties to ensure realism like a real social network: correlated attribute values (see Section 2.6.1), spiking trends (as shown in Figure 2.3) and structure correlations, for example, persons who study at the same university are more likely to become friends. Other graph network generators typically fail to capture how nodes are connected and therefore do not show structural correlations (11, 44). SNB uses a choke-point-based design to stimulate technological advances, optimizing the performance of the choke-points targets (1).

The LDBC SNB Interactive workload covers numerous features that are not directly related to graph data management but are considered important in mature data processing systems. These include: 1) loading large-scale data sets, 2) supporting date- and time-related features, such as correct handling of daylight saving and leap seconds[1], providing functions to extract parts of a timestamp, 3) supporting Unicode strings and nested data structures. Moreover, the workload requires ACID-compliance, see Section 2.6.7.

[1] About Leap Seconds: `https://www.rfc-editor.org/rfc/rfc7164`

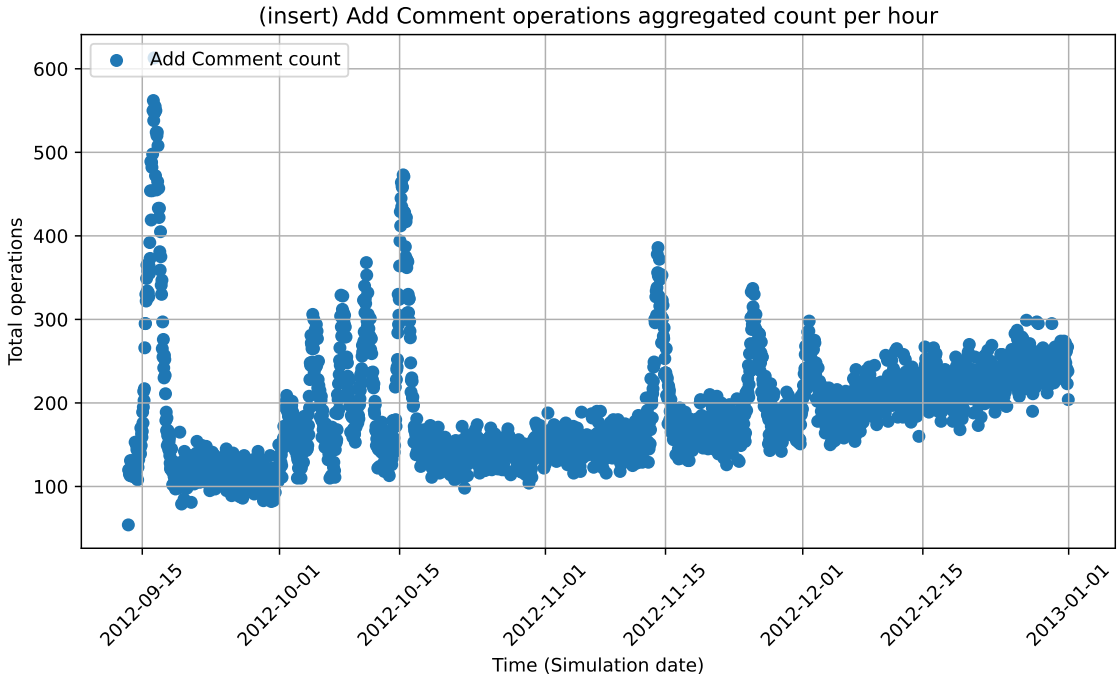

Figure 2.3: Example of a spiking event in the Hadoop Datagen update streams (comment inserts) for Scale Factor 1. Update operations are bucketized per hour according to their simulation timestamp.

The Social Network Benchmark v1.0 consists of multiple software components shown in Figure 2.4. The Datagen including the data output will be discussed in Section 2.6.1. Section 2.6.2 provide a brief introduction to the choke points in Interactive. Next, we discuss in Section 2.6.3 the query templates in general and the substitution parameters from the parameter generation and curation, discussed in Section 2.6.4. Details of the driver, including the workload creation, dependency tracking and execution modes are discussed in Section 2.6.5. Finally, in Section 2.6.6 we discuss the components that an implementation for a SUT consists of, which is the client used by the benchmark driver.

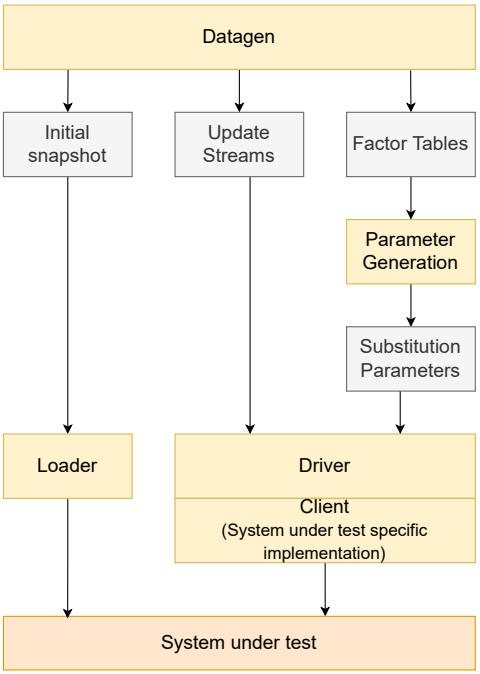

Figure 2.4: Overview of the software components and data artifacts of the Interactive workload. Legend: *Yellow* are the software components, *grey* the data artifacts and *orange* the DBMS tested.

2.6.1 Data and Data Generator (Datagen)

The SNB data generator (Datagen) simulates the users' activity over a specified duration. The data set represents a snapshot of the activity of a social network over three years, simulating the time period between `2010-01-01` and `2012-12-31`. It consists of persons connected through a friendship network and messages that the persons post in message threads on their forums. Figure 2.5 shows the degree distribution for the persons for scale factor 1. Two types of subgraphs are present in the data set: persons form a network of nodes along many-to-many edges whereas the messages, tag classes, and places form

hierarchies. Each person in this data set lives in a city that is part of a country. One may be a member of organizations, companies and universities (18).

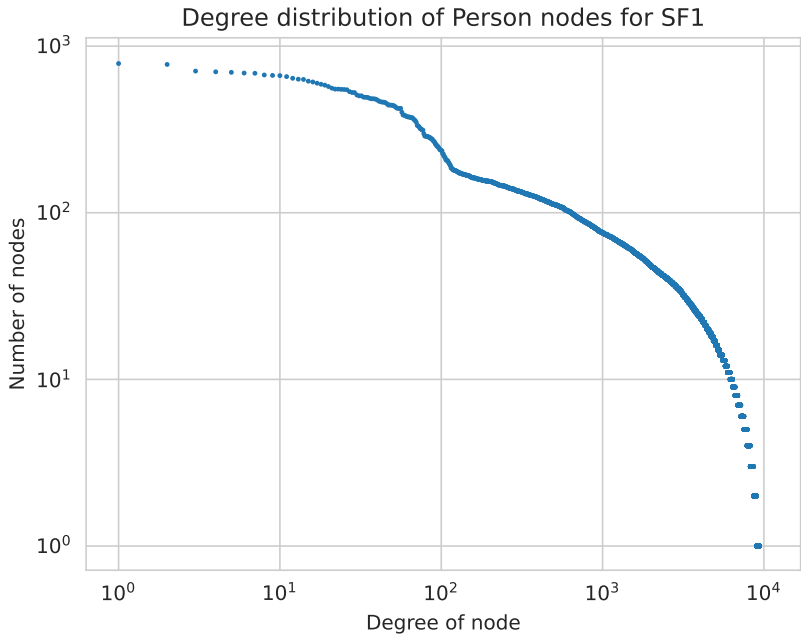

Figure 2.5: Degree distribution of the Person nodes in the SNB graph for Scale Factor 1.

The social network contains correlated attribute values, which also occur in real graphs. These correlations can influence the complexity of algorithms. As such, taking correlations into account in query optimization will likely lead to faster and better query plans in a database system. An example correlated attribute value is the first name a person has depending on the location of that person. Table 2.2 shows the frequency of first names for Germany, China and Spain. The benchmark specification (18, Section 3.3.2) gives a detailed overview of attribute value correlations.

Figure 2.6 shows the schema of the graph. The schema shows how the persons interact with each other on the social network: through friendships, modeled with the knows edge, and interactions using messages by sharing content such as images or text, replying to other messages, and giving likes.

The Interactive workload v1.0 uses a data generator based on Hadoop, referred to as the "Hadoop Datagen" in the following. In 2020, the data generator was migrated to Spark to

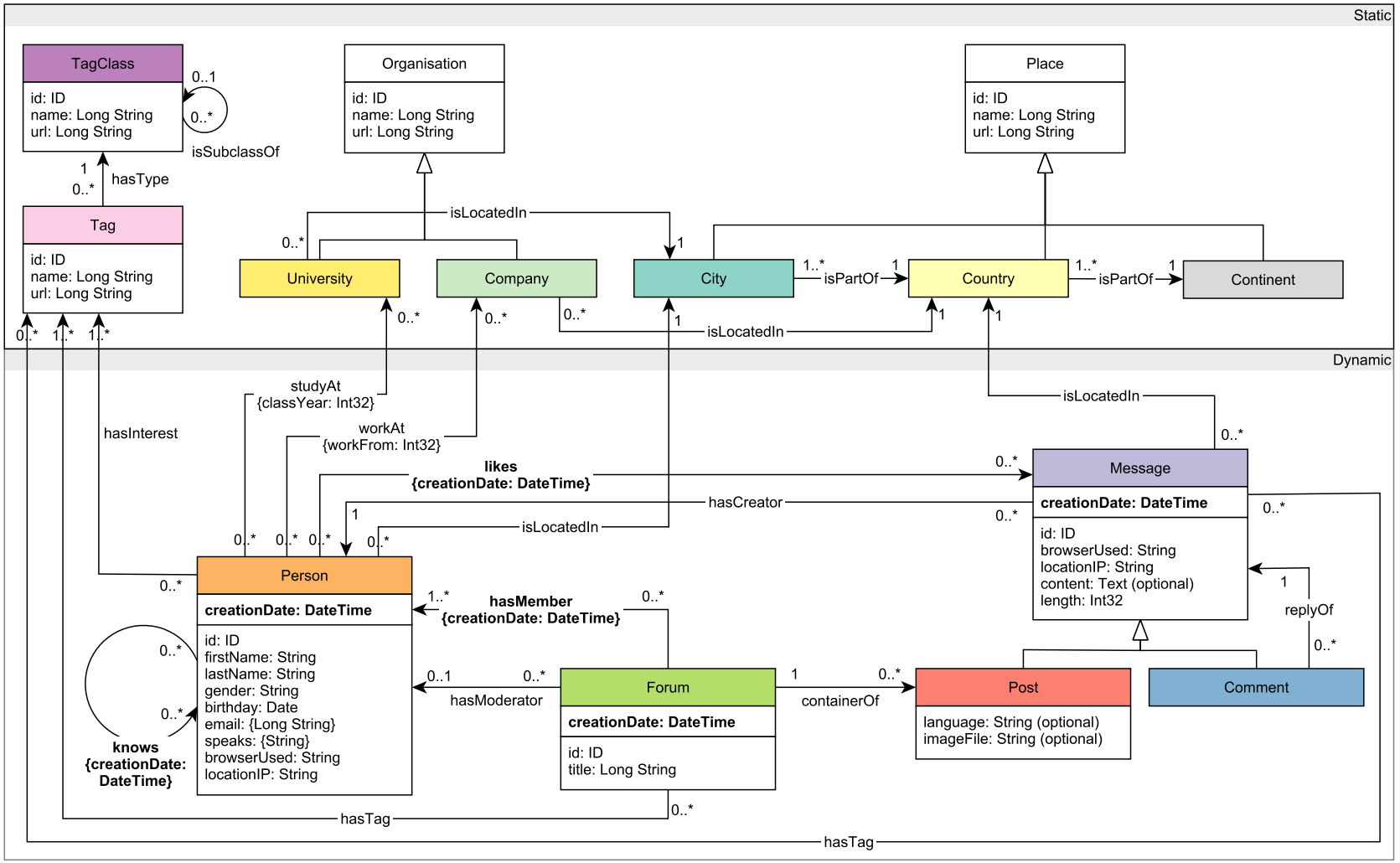

Figure 2.6: UML class diagram-style depiction of the LDBC SNB graph schema. (From the LDBC SNB specification (1))

FirstName	Count
Karl	194
Fritz	173
Hans	171
Wolfgang	156
Rudolf	153
Walter	140
Franz	121
Otto	108
Wilhelm	96
Paul	79

FirstName	Count
Yang	974
Chen	957
Wei	847
Lei	821
Jie	781
Jun	747
Li	552
Hao	507
Lin	422
Peng	419

FirstName	Count
Jose	58
Manuel	54
David	51
Antonio	47
Francisco	33
Carlos	28
Pedro	24
Luis	24
Rafael	24
Javier	23

Table 2.2: Top 10 firstnames for persons from the SF10 Spark Data set with location Germany (left), China (middle) and Spain (right).

increase portability and scalability. This version, referred to as the "Spark Datagen", introduced numerous changes, including changes in the generated CSV schemas and support for serializing files in the compressed Parquet format (21). Moving from Hadoop, disk-based, to Spark, memory-based, allowed for better use of available memory (59). The SNB Interactive v1.0 uses the data generated by the Hadoop Datagen[1], which limits its scalability to SF1,000, while the Business Intelligence workload uses the Spark Datagen[2] and it is able to scale up to SF30,000. Appendix E explains how the data sets were configured for up to SF30,000. Table 2.3 show the number of persons in the graph per scale factor. One of the key objectives of this thesis is to make use of the Spark Datagen's improved scalability in SNB Interactive.

SF	1	3	10	30	100	300	1,000	3,000	10,000	30,000
numPerson	10,620	25,870	70,800	175,950	487,700	1,230,500	3,505,000	9,232,000	27,200,000	77,000,000

Table 2.3: Number of persons in the graph for a given scale factor.

The Spark data generator produces several outputs:

- **Initial snapshot:** the data generated during the simulation before the cut-off date.
- **Factor tables:** tables containing statistics and parameters from the graph, used as input for the parameter generation. Section 2.6.4 will elaborate on the use of the factor tables.
- **Update streams:** data used for the update (and delete) queries. Section 2.6.5 will elaborate on the use of the update streams.

[1] `https://github.com/ldbc/ldbc_snb_datagen_hadoop`
[2] `https://github.com/ldbc/ldbc_snb_datagen_spark`

- **Raw data:** Parquet files containing all the generated temporal graph data.

Both Spark and Hadoop Datagen simulate a graph over three years. However, the cut-off between the initial snapshot and update streams differs: Spark datagen uses 97% of the data for the initial snapshot and 3% for the update streams while Hadoop has a 90%/10% split. Datagen provides the parameters for the update operations: the update streams contain the parameters for the entities and edges.

Table 2.4 shows an overview of update queries present in Interactive v1.0. Each update query comes with a creation timestamp: this timestamp is used by the driver to schedule an update operation during the benchmark.

#	**Update operation**	**Description**
1	Add Person	Add a Person (node) to the graph
2	Add like to Post	Add an edge between a Person and Post denoting a like
3	Add like to Comment	Add an edge between a Person and Comment denoting a like
4	Add Forum	Add a Forum (node)
5	Add Forum Membership	Add an edge between a Person and a Forum
6	Add Post	Add a post (node) in a forum
7	Add Comment	Add a comment (node) in a forum (can be a reply to other comment or reply to post)
8	Add Friendship	Add an edge between two Person nodes

Table 2.4: Overview of update queries used in Interactive v1.0.

In Interactive v1.0, the user needs to specify the number of write threads before generating the data using the Hadoop Datagen. It exports the update stream files partitioned using a Round-Robin strategy. For example, if a user requests 32 write threads, the data generator will split the person event CSV file into 32 files and the forum CSV files into 32 files, resulting in 64 files. Meanwhile, the Spark Datagen generates the update stream files in CSV files stored separately by type, batched per day. The number of CSV files can differ depending on the number of partitions set for Spark. The Spark Datagen additionally outputs update streams for delete operations.

The size of the network depends on the scale factor used. For example, a scale factor of SF300 results in 1,230,500 persons and the total network data has a size of approximately 300 GiB when serialized as CSV files.

2.6.2 Choke Points

The LDBC SNB Interactive workload is designed by choke points: a choke point is an aspect of query execution or optimization that is currently a technical challenge for DBMSs (18). There are multiple types of choke points: *Aggregation performance*, *Join Performance*, *Data Access Locality*, *Expression Calculation*, *Correlated Sub-Queries*, *Parallelism and Concurrency*, *Language features*, *Update operations* and *Graph specific* operations. The full details of all choke points are provided in (1). In this section, we discuss some of the graph-specific choke points.

Unweighted shortest paths: test the ability to compute the distance values between a node and a set of nodes.

Composition of graph queries: tests the support of composable graph queries in the database system. Composable graph queries are multiple graph queries where the first one defines a subgraph which is then passed to the next query.

Reachability between disconnected components: tests the path-finding queries whether in a single-source single-destination search the database system can assert quickly if a path does not exist, for example, by using bidirectional path-finding algorithms or running connected components as a preparatory step.

2.6.3 Query Templates

The SNB workloads use pre-defined query templates to issue queries against the data set. A query template is an expression with substitution parameters, also known as parameter bindings, that are replaced by the workload generator with parameters for the data set to make the benchmark results understandable. These parameters need to be selected carefully such that the benchmark queries have stable behavior when the data set contains skewed data distributions and data correlations (28).

An example of a query in the Interactive workload is Query 3, pattern shown in Figure 2.7, where given a `personId`, it finds persons that are friends and friends of friends that made posts/comments in both given countries in a given time interval. The query template is given in Listing 2.5. In this example the query template has 5 parameters that need to be substituted by the workload generator: `personId`, `countryXName`, `countryYName`, `startDate` and `durationDays`.

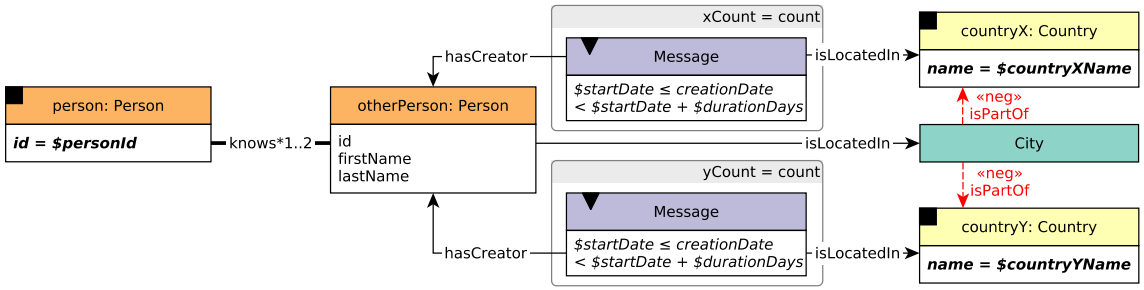

Figure 2.7: Query template Query 3 (1). The dashed red lines note negative conditions, i.e., these edges must not exist in the graph.

```
SELECT
    Person.id AS otherPersonId, Person.firstName AS otherPersonFirstName,
    Person.lastName AS otherPersonLastName, ct1 AS xCount,
    ct2 AS yCount, totalcount AS count
FROM (
    SELECT Person2Id FROM Person_knows_Person
    WHERE Person1Id = :personId
    UNION
    SELECT k2.Person2Id FROM Person_knows_Person k1, Person_knows_Person k2
    WHERE k1.Person1Id = :personId AND k1.Person2Id = k2.Person1Id
      AND k2.Person2Id <> :personId
) friend, Person, City, Country,
(
    SELECT chn.CreatorPersonId, ct1, ct2, ct1 + ct2 AS totalcount
    FROM (
        SELECT CreatorPersonId AS CreatorPersonId, count(*) AS ct1
        FROM Message msg, Country
        WHERE LocationCountryId = Country.id AND Country.name = :countryXName
          AND msg.creationDate >= :startDate
          AND msg.creationDate < (:startDate + INTERVAL '1 days' * :durationDays)
        GROUP BY CreatorPersonId
    ) chn, (
        SELECT CreatorPersonId AS CreatorPersonId, count(*) AS ct2
        FROM Message msg, Country
        WHERE LocationCountryId = Country.id AND Country.name = :countryYName
          AND msg.creationDate >= :startDate
          AND msg.creationDate < (:startDate + INTERVAL '1 days' * :durationDays)
        GROUP BY CreatorPersonId
    ) ind
    WHERE chn.CreatorPersonId = ind.CreatorPersonId
) cpc
WHERE friend.Person2Id = Person.id AND Person.LocationCityId = City.id
  AND City.PartOfCountryId = Country.id AND Country.name <> :countryXName
  AND Country.name <> :countryYName AND friend.Person2Id = cpc.CreatorPersonId
ORDER BY totalcount DESC, Person.id ASC
LIMIT 20;
```

Listing 2.5: Query template LdbcQuery3 formulated in SQL from the Interactive implementation

2.6.4 Parameter Curation

In order to have interpretable runtime measurements of the performance of the system, parameter curation is applied to the selection of query parameters. Parameter curation transforms the behavior of benchmarks from unpredictable, often multi-modal distributions, into an approximate normal distribution for the same query template (28). Without parameter curation, data sets with skewed data distributions and value correlations show unstable behavior: the same benchmark query can have runtimes with high variance.

The factor tables contain parameters extracted from the generated data used by parameter curation, which selects parameters with similar properties, e.g., days with roughly similar frequencies of messages created and countries with similar message frequencies. Table 2.5 shows examples of statistics from three factor tables.

creationDay	numMessages
2012-09-12	76,210
2012-09-13	75,773
2012-09-14	79,874
2012-09-15	110,976
2012-09-16	130,475
2012-09-17	92,977
2012-09-18	78,218
2012-09-19	77,079
2012-09-20	77,684

Country	numMessages
Germany	940,725
South Africa	251,320
Chile	253,417
Netherlands	222,750
England	226,764
India	4,822,947
Japan	898,030
United States	867,166
Spain	277,857

Language	frequency
en	1,220,272
zh	209,173
es	123,670
fr	92,034
ar	68,651
ru	57,005
de	48,907
ur	45,498
pt	38,867

Table 2.5: Example of three factor tables from Datagen: (left) daily bucketized number of messages, (middle) number of messages created per country, (right) number of messages grouped by language.

Parameter curation in SNB v1.0 The parameter curation in v1.0 uses an algorithm that first searches for parts in the factor tables with the smallest variance across for each column, called *windows*. For each column, the windows are then merged with windows also having the smallest variance. This continues until there are no windows left or the last column is reached. The output of the parameter generation is a file for each query containing the substitution parameters.

2.6.5 Operations in the Interactive Workload

The Interactive workload targets DBMSs capable of processing graph workloads that combine transactional updates with query capabilities. There are three classes of queries: *transactional update queries*, consisting of the insert operations pre-generated by the SNB

data generator, e.g., update operations are the addition of friendships, forums, posts/-comments, and likes to a post/comment. *Complex read-only queries* retrieve information about the social environment of a given person (one-, two, or three-hop neighborhoods) and perform path-finding. Thirdly, there are simple *short read-only queries* that perform lookups on information about someone's profile and post information. Systems are expected to compete on achieving a high *Time Compression Ratio* (TCR^{-1}), which compresses/stretches the duration between the scheduled creation timestamp to increase (or decrease) the operation rate (1, p9). For example, a TCR^{-1} of 10 means that the system can execute the workload 10× faster than it was modeled to occur, namely, the simulation time.

2.6.5.1 Tracking Dependencies

To allow scalable benchmark execution, a transactional workload must be partitioned into streams that are issued concurrently against the system under test (SUT) (18). However, the update streams containing the insert queries are not trivial to partition since update operations may depend on each other: to create a friendship, the two persons must exist in the network. Likewise, one can only reply to a message once posted. While the messages can be partitioned, e.g., by grouping the dependent messages, the person network with the friendship edges cannot. To enable concurrent update streams that have dependencies, the driver tracks the dependencies based on the operation type and its scheduled simulation time. Every operation has a T_{due} time representing the simulation time at which that operation is scheduled to execute. Each update operation belongs to none, one, or both of two types of sets: *Dependencies* and *Dependents* (18, Section 4.2). *Dependencies* are operations that introduce a dependency in the workload, for example, creating a person with corresponding information in the network. *Dependents* contain operations dependent on at least one other operation in the *Dependencies* sets. Therefore, operations in the dependent set cannot be executed until the corresponding operation from the dependencies is executed. The Datagen however ensures that there is a minimum time duration, T_{safe}, between dependent operations to reduce synchronization overhead in the driver when executing operations. The driver then only needs to check every T_{safe} time if an operation can be executed. By default, T_{safe} is set to 10 seconds in the simulation time.

The driver tracks the latest point that every operation with a T_{due} lower or equal to this time is guaranteed to have completed execution (18) by maintaining a completion time service. In this service, the *Dependencies* operations' time stamps are registered first in

the Initiated Times (IT) set and, after completion, removed from IT and added to the Completion Times (CT). Timestamps can only be added to the IT in a monotonically increasing order but can be removed in any order. Each update operation stream tracks the initiated and completion times. It does this by writing the start and completion times, which is monitored by the completion time service. The simulated timestamp is multiplied with the TCR^{-1}, compressing/stretching the timestamps for the scheduled start times. If during the execution of the workload the driver waits before executing the next operation due to the dependent time, this is a sign that the SUT cannot keep up with the workload, hence, a lower TCR^{-1} should be chosen.

2.6.5.2 Workload Creation

To create the workload, the user specifies the properties required for loading and execution, summarized in Listing 2.6. The driver loads the update stream files as separate workload streams. Once loaded, the driver reads the lowest start time of all the streams to determine the start time of the workload. Next, the driver schedules the complex read queries depending on the update interleave and the frequency of each query type. The *update interleave* denotes the average time between each update operation and is calculated as follows:

$$\text{update interleave (in ms)} = \frac{(T_{end} - T_{start})}{\text{total operations}} \tag{2.1}$$

```
1 thread_count=1
2 time_compression_ratio=0.05
3
4 warmup=6000 # Number of operations during warmup
5 operation_count=30000 # Number of operations during benchmark
6
7 ldbc.snb.interactive.scale_factor=10
```

Listing 2.6: Example of benchmark properties

The frequency of each query is different depending on the scale factor specified by the user: the rationale behind the frequencies is to balance the total runtime of each complex query type such that the total runtimes are approximately the same per query. The frequencies make each query equally important for the benchmark score and it ensures that the choke points are stressed sufficiently. To calculate the time between instances of a complex query of the same type, we use the following formula:

$$\text{query interleave (in ms)} = f_{query} \times \text{update interleave} \tag{2.2}$$

The driver schedules the complex read queries by setting the start time of the update streams and increasing the start time monotonically using the query interleave. Figure 2.8 shows an example of the scheduling of the queries. In the Interactive v1.0 workload, the user needed to provide the correct frequencies per scale factor. To improve usability, we changed this to include the frequencies in the driver (since these are after they are specified not changed).

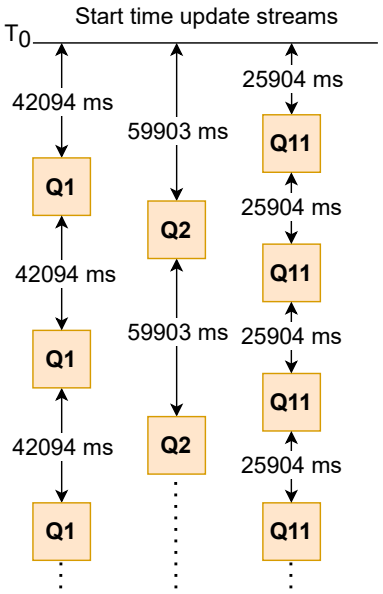

Figure 2.8: Example of scheduling of complex read queries with an update interleave of 1619 ms and frequencies Query1=26, Query2=37 and Query11=16, see Equation 2.2

The short read queries are not scheduled during the initialization of the workload. During the execution of the benchmark, the results of some complex and short read queries provide the starting points for the short read queries. Appendix A shows the queries that give input parameters to the short read queries. The results are added to a buffer containing person IDs and message IDs used after execution. The short reads are executed after each eligible complex read operation. The amount of short reads executed depends on the result size of that long read operation, and the type of short reads depends on whether the result contains person IDs and message IDs. The short reads use an interleave that equals the update interleave multiplied by the TCR^{-1}. After each execution, the probability that a short read executes diminishes with the *short read dissipation factor*. This factor determines the threshold number when a new short read executes after a random "coin-toss".

2.6.5.3 Benchmark Execution

After the initialization of the workload, the driver executes the workload in two phases: a warmup phase and a measurement window. The warmup phase puts the system in a steady state as it would behave in a normal operating environment and is, therefore, not part of the measurement. Both phases depend on the number of operations the user sets in the properties file. To achieve a two-hour run, the user needs to set the number of operations by using the expected throughput, for example, with a throughput of 3,000 operations/s:

$$\text{operation count} = \textit{throughput} \left[\frac{1}{\text{s}}\right] \times \textit{duration} \text{ [s]} \tag{2.3}$$

$$21{,}600{,}000 = 3{,}000 \times 7{,}200 \tag{2.4}$$

The execution of the workload by the driver requires a client implementation of the SUT. The driver provides interfaces that a user implements to handle the workload. The client implementation should implement execution handlers for each workload query, parsers for each query's results, and a class that handles connections with the SUT.

2.6.5.4 Cross-Validation

To determine whether the SUT produces correct results, the driver offers two modes: the creation of validation parameters and validation using results from another system. With the creation of the validation parameters, the result of each query is stored together with its substitution parameters in a CSV file. This CSV file can then be used against other DBMSs using the validation mode to cross-validate whether the results are the same. The number of validation parameters generated needs to be sufficiently high to cover multiple instances of all queries. To create the validation parameters, sequential execution is used. If the creation of validation parameters would be done in a multithreaded concurrent run, the query answers may be different due to different execution orders and would therefore not guarantee deterministic results.

2.6.6 Implementation

To run the benchmark, an implementation has to be created for the system under test. An implementation consists of the following elements:

- **Query Templates:** The query templates are specific to the implementation's supported language and features. These can, for example, be written declaratively in SQL or imperatively in a programming language like C++.
- **Operation Handler:** The operation handler handles the execution of a query. An implementation can have multiple operation handlers.
- **Connection Client:** The connection client that the interactive driver operation handler uses.
- **Converters:** Converters are used to change parameters to implementation-specific formats. This can for example be a datestamp in a specific format.

2.6.7 ACID Compliance

The verification of ACID compliance in the benchmarking process is required to enable a fair comparison between systems, since the performance benefits of weaker safety guarantees are known, including potentially incorrect query results (70). ACID stands for Atomicity, Consistency, Isolation, and Durability. These properties ensure that a transaction always leaves the database in a valid state:

- **Atomicity:** ensures that either all operations in a transaction are performed or none.
- **Consistency:** the database remains consistent during the execution of transactions in isolation. Additionally, each transaction takes the database from one consistent state to another.
- **Isolation:** The DBMS must ensure that each transaction is unaware of other transactions executing concurrently. There are several isolation levels: *Serializability*, which is the most strict isolation level, *Read Committed*, which guarantees that any data read is committed at the time it is read by the `SELECT` statement in the same transaction. Lastly, *Snapshot Isolation*, which guarantees that all reads in a transaction will see a consistent snapshot of the database.
- **Durability:** After a successful transaction, the changes in the database persist, even in case of a system failure.

The ACID tests are separate from the Interactive driver and are outside the scope of this thesis.[1]

[1] `https://github.com/ldbc/ldbc_acid`

Part I

SNB Interactive v2.0

3

Design & Implementation

This thesis aims to add support for deletions and scale factors above SF1,000 in the Interactive workload, approximate the number of persons for scale factors above SF3,000, and to create a reference architecture for a distributed driver. Implementing deletes requires extensive changes in the driver and the parameter generation: the driver needs to be aware of temporal substitution parameters and support the data produced by the Spark Datagen, described in Section 2.6.1. In addition, we introduce new variants for Query 3, 13, and 14, where Query 14 comes with a new definition compared to v1.0. This chapter will describe the key components of the upgraded driver and the changes introduced when migrating from the Hadoop data set to the Spark data set. These changes affect the implementations repository, where reference implementations are updated to support the new data set, naming convention, CSV schemas, and queries. Figure 3.1 gives an overview of the components with the changes highlighted in blue.

3.1 Overview

This section describes the changes in the Interactive workload's components. Section 3.2 discusses the changes in the driver required to support the new Spark Datagen containing the delete operations and support for larger scale factors. In Section 3.3, we discuss the time-aware scalable parameter curation, enabling parameter selection for entities inserted and deleted during the execution of the benchmark, as well as a new method to curate path-query parameters. The different sections come together in Section 3.4 when we discuss updating the driver.

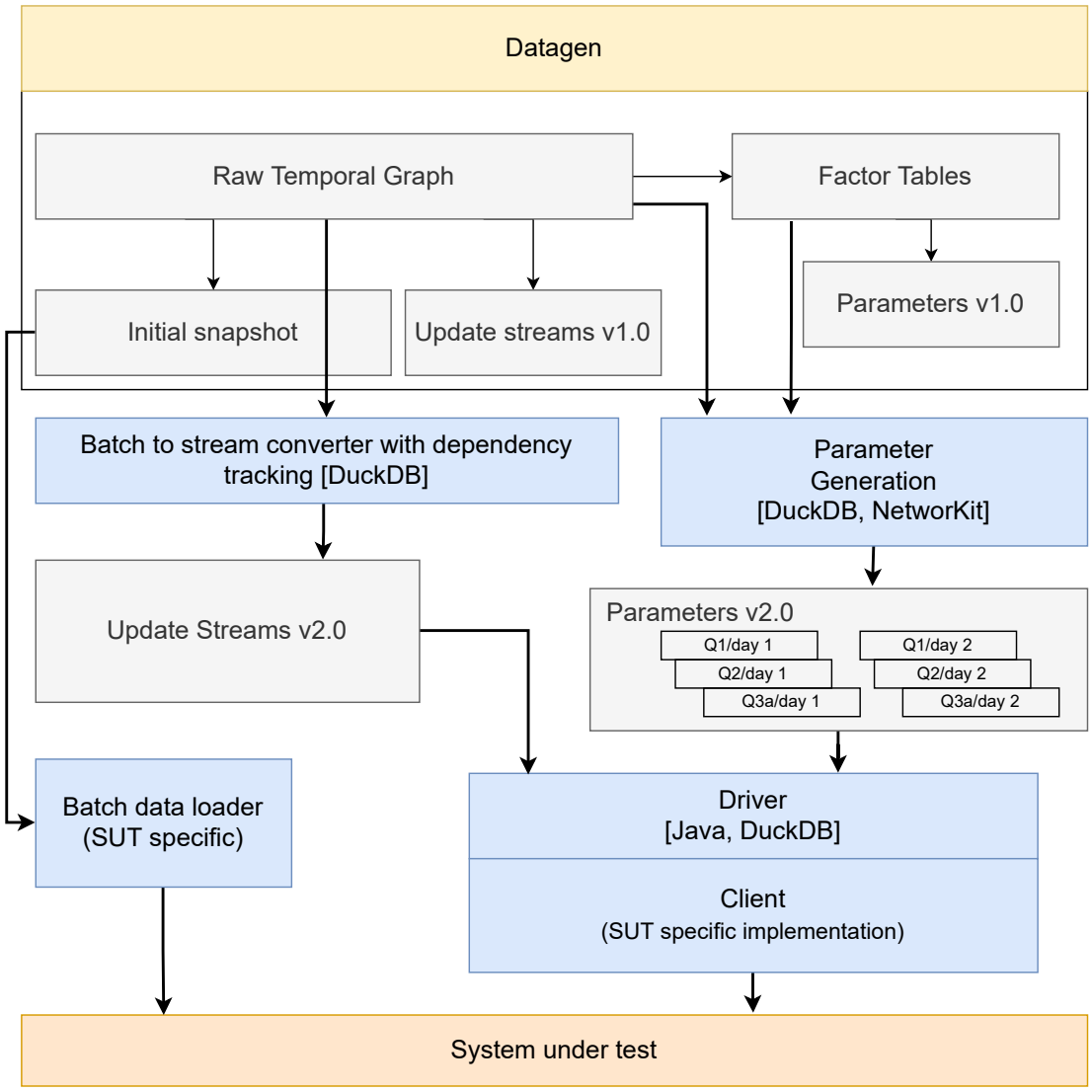

Figure 3.1: Schematic overview of key changes in the Interactive benchmark. The changes are colored *blue*. The data artifacts are colored *grey*. Components part of LDBC SNB but not changed in this thesis are marked *yellow*. Components not part of LDBC SNB are marked *orange*.

3.2 Migrating the Driver from the Hadoop Datagen to the Spark Datagen

We migrate the driver from using data sets produced by the Hadoop Datagen to the Spark Datagen to add support for deletions and larger scale factors. As covered earlier, the Spark Datagen includes temporal entities with lifespan attributes for deletions. This feature was implemented in the LDBC Spark Datagen by Waudby et al. (71). The entities and edges have a creation date and deletion date, as shown in Figure 3.2.

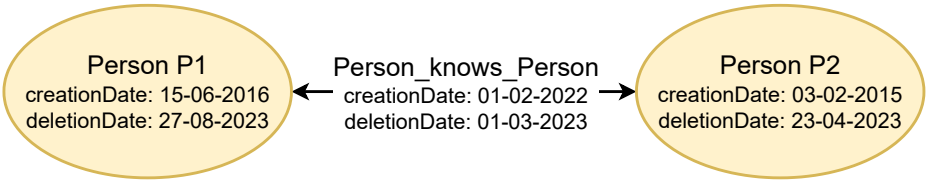

Figure 3.2: Example of lifespan management of an entity and edges. Each node and vertex has a creationDate and deletionDate specified.

The Spark Datagen exports the update streams in a different format than the Hadoop Datagen. Hadoop exports the update streams depending on the number of write threads, p_w, specified by the user, resulting in p_w update stream files with person updates and p_w forum files containing the insert queries 2 to 8 (Table 2.4). Spark exports the update streams in a separate folder, depending on whether it is a delete or insert operation. However, the update streams of the Spark Datagen lack the dependent time column required by the interactive driver to track the dependencies during execution. Hence, as part of this thesis, we introduce the dependent time columns to the update streams produced by the Spark Datagen.

3.2.1 Creating the Dependent Time Column

The dependent time column contains the time of the entity (dependency) to which the update is dependent. In the Hadoop Datagen, the dependency time was determined incorrectly by basing the dependency time on the creation date of a Person entity or the author of a message. However, this approach is only correct in some cases since updates regarding comments, posts, and forums can have a dependent time that is more recent than the creation date of a person. However, Figure 3.3 shows an example of a comment with dependencies where the creation date of the replied comment is more recent than the

creation date of the person.

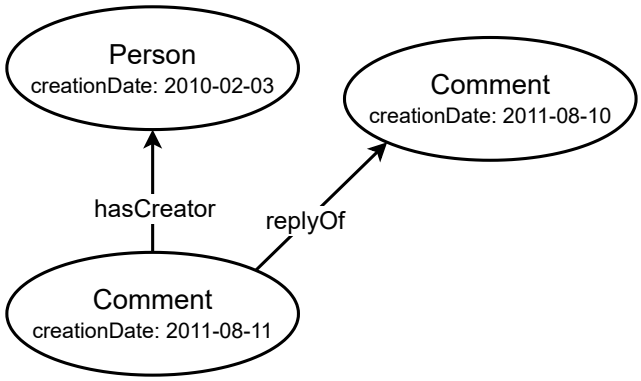

Figure 3.3: Example of an entity with dependencies.

To select the dependency time for each entity, we define the dependencies for each update query, shown in Table 3.1. Note that only adding a person to the graph has no dependencies. Other insert queries rely on the creation date of the person or entity to which it has an edge. Delete operations depend on the entity's creation date or one of the end nodes when a delete query touches an edge. In the example of the message, a message has an author, but the message can be a reply to another message, having a total of two dependencies.

Type	**Query/Entity**	**Dependency**
Insert	AddPerson	No dependency
Insert	AddLikeToPost	Greatest creationdate between Person and Post liked
Insert	AddLikeToComment	Greatest creationdate between Person and Comment liked
Insert	AddForum	Person creationdate (moderatorPerson)
Insert	AddForumMembership	Greatest creationdate between Person and Forum
Insert	AddPost	Greatest creationdate between Person and Forum Post belongs to
Insert	AddComment	Greatest creationdate between Person and Comment replied to
Insert	AddFriendship	Greatest creationdate of both persons
Delete	DeletePerson	Creationdate of Person
Delete	DeletePostLike	Greatest creationdate of Person, Post
Delete	DeleteCommentLike	Greatest creationdate between Person and Comment liked
Delete	DeleteForum	Creationdate of Forum
Delete	DeleteForumMembership	Creationdate of Person or Forum
Delete	DeletePost	Creationdate of Post
Delete	DeleteComment	Creationdate of Comment
Delete	DeleteFriendship	Creationdate of knows edge between Person1 and Person2

Table 3.1: Update query with the attribute(s) used to determine dependency time.

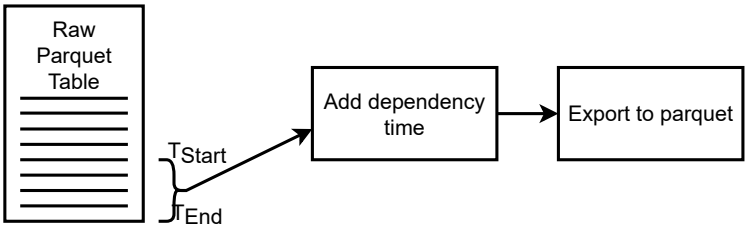

Figure 3.4: Creation of the update streams. The rows from the Parquet file from the time split T_{Start} are extracted and afterward the dependency time is added depending on the entity. The result is exported to Parquet.

3.2.2 Exporting Update Streams

We create the update streams using the raw Parquet files (see Section 2.6.1), which contain all the information about the graph. The motivation to use the Parquet files instead of merging the CSV files is that by using the Parquet files directly, we combine the extraction of the dependency time with the creation of the update streams and fewer files need to be loaded. To select all entities that are part of the update stream, we use the split defined in the Spark Datagen: 97% of the simulation time in epoch milliseconds is in the initial snapshot, and 3% is for the update streams.

To set the dependency time, we must select for each dependent entity the most recent creation date of the dependencies, which we can select from the raw Parquet files using DuckDB (49), running embedded (in-process) in the benchmark driver. DuckDB enables us to directly query Parquet files using SQL. The process is shown in Figure 3.4. The addition of the dependency time can require joins with multiple large tables, for example, the `Comment` table and `Comment_hasTag_Tag`, potentially leading to out-of-core operations for larger scale factors. To mitigate this, we batch the creation of the update streams, creating an update stream file per batch. After all the rows have been loaded, the batches are merged into a single Parquet file.

Listing 3.1 shows an example of the SQL query selecting the comments for the insert operations together with the dependency time. This is batched by exporting a `Comment-*.parquet` file per batch. In contrast, Listing 3.2 shows the SQL query for selecting the comments for the delete operations which does not require batching. The results are 16 Parquet files containing the update streams, one for each update operation type.

```
COPY (
    SELECT
        c1.creationDate,
        greatest(Person.creationDate, c2.creationDate) AS dependencyTime,
        c1.id, c1.locationIP, c1.browserUsed, c1.content,
        c1.length, c1.CreatorPersonId, c1.LocationCountryId,
        c1.ParentPostId, c1.ParentCommentId,
        string_agg(DISTINCT Comment_hasTag_Tag.TagId, ';') AS tagIds
    FROM Comment c2, Person, Comment c1
    LEFT JOIN Comment_hasTag_Tag
           ON Comment_hasTag_Tag.CommentId = c1.id
    WHERE c1.creationDate > :start_date_long
      AND c1.creationDate < :end_date_long
      AND Comment_hasTag_Tag.creationDate > :start_date_long
      AND Comment_hasTag_Tag.creationDate < :end_date_long
      AND c1.ParentPostId IS NULL AND c2.id = c1.ParentCommentId
      AND c1.CreatorPersonId = Person.id
    GROUP BY ALL
    ORDER BY c1.creationDate
)
TO ':output_dir/inserts/Comment-:index.parquet' (FORMAT 'parquet');
```

Listing 3.1: Example of a SQL query to determine the dependency time for a comment insert using DuckDB.

```
COPY (SELECT deletionDate, creationDate AS dependentDate, id
     FROM Comment
     WHERE deletionDate > :start_date_long
       AND explicitlyDeleted = true
     ORDER BY deletionDate ASC)
TO ':output_dir/deletes/Comment.parquet' (FORMAT 'parquet');
```

Listing 3.2: Example of a SQL query to determine the dependency time for a comment delete using DuckDB.

3.3 Time-Aware Scalable Parameter Curation

Using the Spark Datagen has the effect that the format of the factor tables is different: v1.0 uses CSV factor tables from Hadoop Datagen for the parameter generation and curation whilst the Spark Datagen uses Parquet factor tables. In addition, the Hadoop Datagen factor tables only include parameters and statistics that were included in the initial snapshot: inserted entities where not taken into account. Therefore, the parameter generation in Interactive v1.0 (see Section 2.6.4) did not have to take the temporal aspects of the workload into account. The parameters generated in Interactive v1.0 where then used during the entire simulation time of the benchmark.

For v2.0, the factor tables exported from the Spark Datagen contain parameters with statistics using the entire simulation time, including nodes and edges that are inserted and deleted (part of the update streams). This however leads to problems when selecting pa-

rameters since the parameter is only valid for *some* time since nodes and edges can be added and deleted throughout the simulated time affecting the statistics. Additionally, parameters present in the factor table may not exist at the start of the benchmark or are removed during the benchmark. To mitigate this problem, the new parameter generator selects parameters in daily batches based on the creation and deletion time of the parameter. The factor tables are loaded using DuckDB to enable us to select parameters using SQL queries.

3.3.1 Selecting Factor Tables

For the parameter selection, we determine for each query which factor tables are required. Selecting the factor tables should be done to approximate the intermediate cardinalities for a query given a parameter. We provide the description of each query from the specification (1) together with the pattern, then we elaborate on which factor tables to use. We highlight some queries to show the design and selection process. The remainder of the queries is shown in Appendix B.

Query 1

> *Given a start Person with ID* `$personId`*, find Persons with a given first name (*`$firstName`*) that the start Person is connected to (excluding start Person) by at most 3 steps via the knows relationships. Return Persons, including the distance (1..3), summaries of the Persons' workplaces and places of study. (1)*

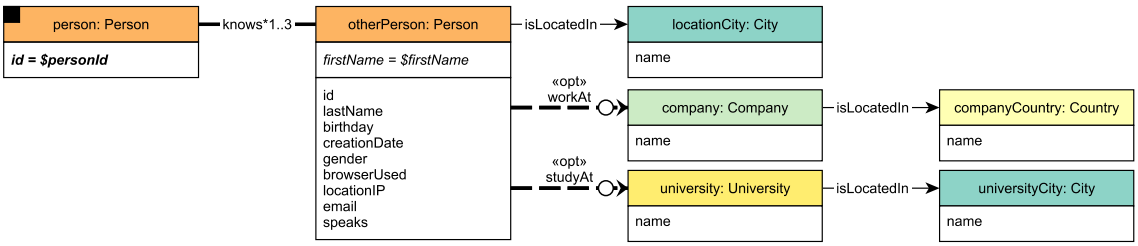

Figure 3.5: Pattern of LDBC SNB Interactive Complex Read 1

For each `$personId` parameter in Query (Figure 3.5), we want a similar number of persons in the third hop. For this, we use the `personNumFriendsOfFriendsOfFriends` factor table to select person IDs with a similar number of friends in the third hop. To select the `$firstName` parameter, we use the `personFirstNames` factor table, containing a list of first names together with the frequency of occurrence in the graph.

Query 2

> *Given a start Person with ID* `$personId`*, find the most recent Messages from all of that Person's friends (friend nodes). Only consider Messages created before the given* `$maxDate` *(excluding that day). (1)*

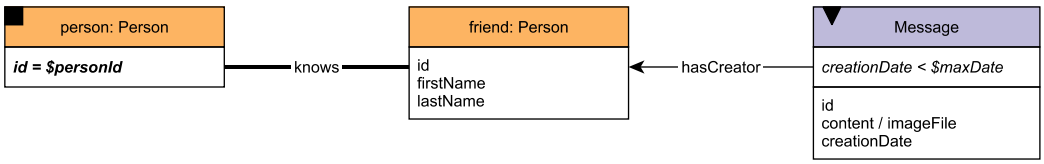

Figure 3.6: Pattern of LDBC SNB Interactive Complex Read 2

Query 2 (Figure 3.6) requires us to know the amount of Messages the friends of the selected person ID with dates before a selected `$maxDate`. There is no direct factor table that can be used to know the amount of Messages for a given date for a given person ID. While we could use the `personNumFriendComments` table to select a person ID with a similar amount of comments made by friends, this only gives us the number of comments during the whole simulation time frame, so depending on the scheduled time it may not be a good approximation. Therefore, we select a person ID based on the number of friends using the `personNumFriends` table, together with the `creationDayNumMessages`, which gives us the number of comments in the graph made for a given date. This lets us select dates where the number of comments have smaller differences.

Query 3a and 3b

> *Given a start Person with ID* `$personId`*, find Persons that are their friends and friends of friends (excluding the start Person) that have made Posts / Comments in both of the given Countries (named* `$countryXName` *and* `$countryYName`*), within [*`$startDate`*,*`$startDate` *+* `$durationDays`*) (closed-open interval). Only Persons that are foreign to these Countries are considered, that is Persons whose location Country is neither named* `$countryXName` *nor* `$countryYName`*. (1)*

Query 3a and 3b (Figure 3.7) will search up to the second hop friends for posts and comments made in a time interval. The posts need to be in locations different than the author of the post. For this query, Interactive v2.0 has two variants: Query 3a, which has two countries that have a high correlation in the friendship network and Query 3b, with anti-correlated countries: countries that do not have a high frequency of friendships between

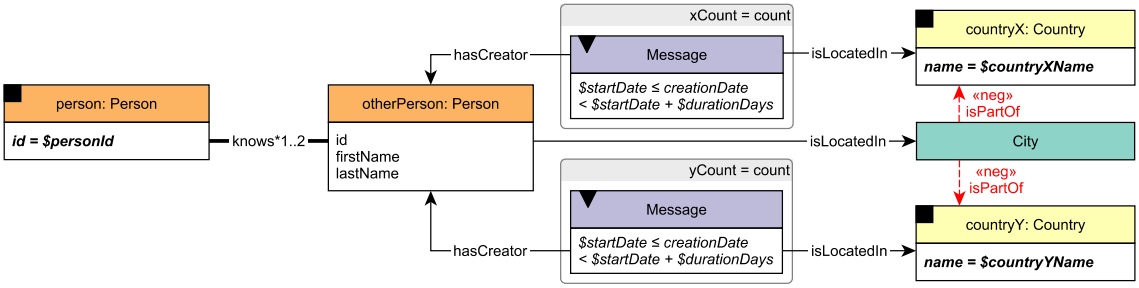

Figure 3.7: Pattern of LDBC SNB Interactive Complex Read 3

persons. To select the countries, we can use the `countryPairsNumFriends` table, which contains the names of country pairs together with the number of friendships between those countries. An example of correlated and anti correlated countries is shown in Table 3.2. For Query 3a, we select countries with high frequencies and for 3b countries with low frequencies. To select a suitable person ID, we can use the `personNumFriendsOfFriends` table which contains the number of friendships in the second hop. Lastly, we use the `creationDayNumMessages` to select a range of dates with a similar amount of messages posted/commented in the network.

Correlated		
CountryXName	**CountryYName**	**Frequency**
Japan	Philippines	2973
Indonesia	Sri Lanka	2993
Egypt	Turkey	3011
England	United Kingdom	2932
Moldova	Ukraine	3023

Anti-Correlated		
CountryXName	**CountryYName**	**Frequency**
Angola	Croatia	1
Angola	Denmark	1
Angola	Hong Kong	1
Angola	Libya	1
Angola	Lithuania	1

Table 3.2: Examples of correlated and anti-correlated country pairs from the countryPairsNumFriends factor table.

Query 4

> *Given a start Person with ID* `$personId`*, find Tags that are attached to Posts that were created by that Person's friends. Only include Tags that were attached to friends' Posts created within a given time interval [*`$startDate`*,* `$startDate + $durationDays`*) (closed-open) and that were never attached to friends' Posts created before this interval. (1)*

For Query 4 (Figure 3.8) we want a person parameter with a similar number of direct friends and a start date with a similar amount of messages created. Therefore, we use the `creationDayNumMessages` factor table to select a suitable date and the `personNumFriends`

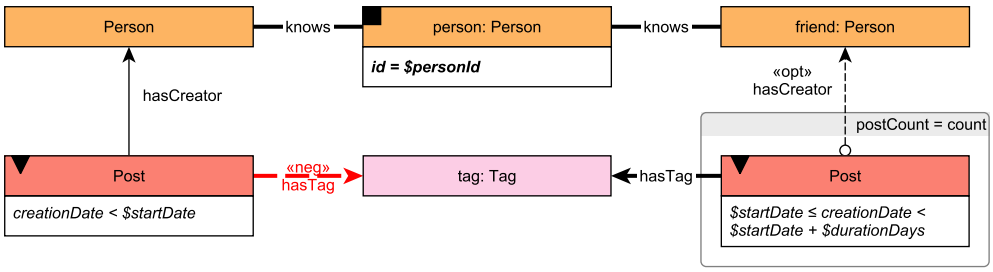

Figure 3.8: Pattern of LDBC SNB Interactive Complex Read 4

table to select a person with a similar number of friends. The number of `durationDays` is generated by generating a series from 1 to 20.

Query 5

> *Given a start Person with ID* ***`$personId`****, denote their friends and friends of friends (excluding the start Person) as otherPerson. Find Forums that any Person otherPerson became a member of after a given date (* ***`$minDate`****). For each of those Forums, count the number of Posts that were created by the Person otherPerson. (1)*

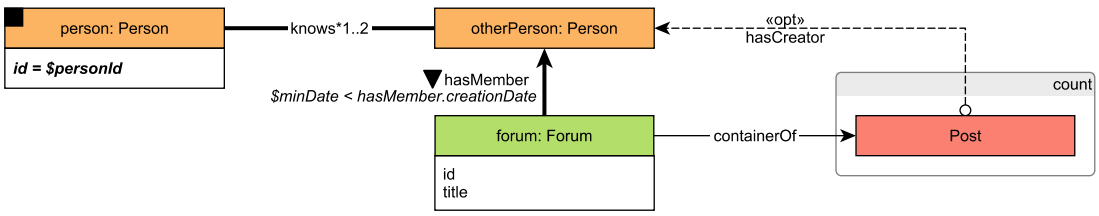

Figure 3.9: Pattern of LDBC SNB Interactive Complex Read 5

In Query 5 (Figure 3.9), the first- and second-degree friendships are taken into account. We use the `personNumFriendOfFriends` factor table to select a person ID with a similar second-degree friendship size. During the query, the forum memberships of each friend and friend of friends are checked to ensure that they are after a certain date. Therefore, we select person IDs with a similar amount of forum counts as well, which are provided in the `personNumFriendOfFriendForums`. PersonIDs are then selected when they occur in both selections.

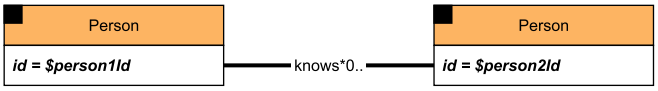

Figure 3.10: Pattern of LDBC SNB Interactive Complex Read 13

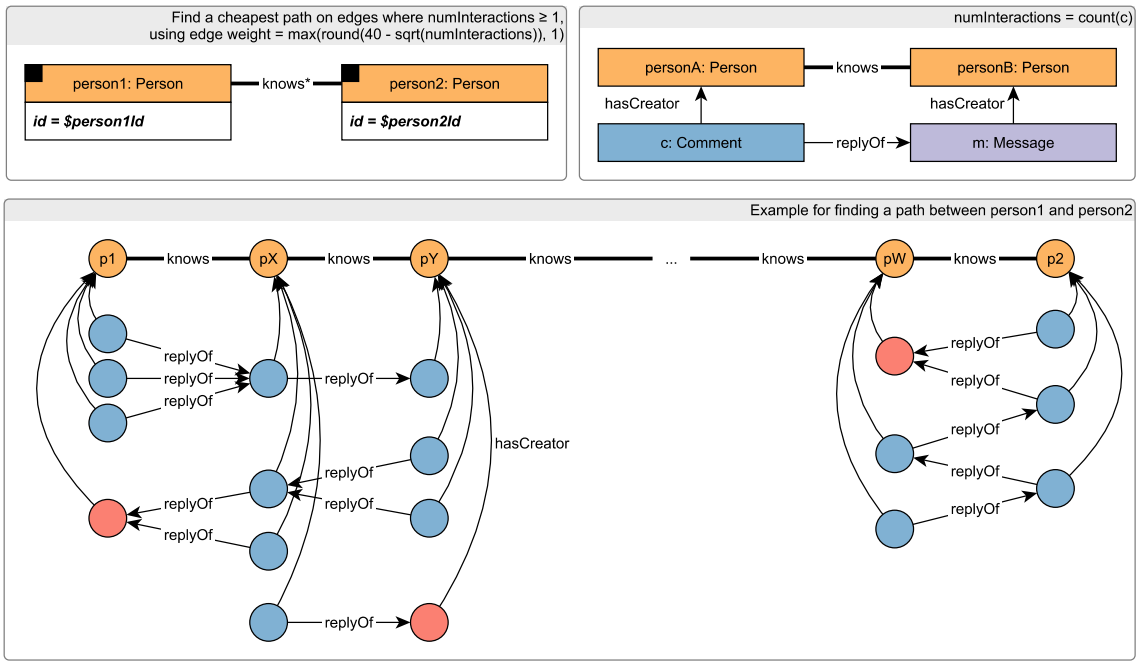

Figure 3.11: Pattern of the new LDBC SNB Interactive Complex Read 14

Query13 and 14

Query 13 (Figure 3.10) and 14 (Figure 3.11) are path-finding queries: they give the shortest/cheapest path between two persons and return the path. Interactive v2.0 introduces variants for both queries: 13a and 14a do not contain a path between the two persons while 13b and 14b *do* contain a path. To select suitable person pairs which have a path, we use the `people4Hops` factor tables (containing pairs of Person nodes who have a 4-hop path between them at some point). In Section 3.3.3 we elaborate on selecting paths from the `people4Hops` factor table.

Query 13a and 14a Query variants 13a and Query 14a do *not* have a path between the two persons. For both queries, we select person IDs from a different component using the `personKnowsPersonConnected` table. This table contains person IDs together with the connected component it is part of and the count of that component: since not all persons are part of the same connected component, we select a person IDs from two different components.

Query 13b For Query 13b, we need to select two person IDs from the `people4Hops` table, which provides us with two person IDs with a path 4 hops away. Additionally, we need to select the person IDs that have similar number of friends in the first and second hop, which the `personNumFriendOfFriends` table provides.

Query 14b With Query 14b, we also need to select two person IDs from the `people4Hops` table. In contrast with Query 13b, which does not use edge weights, we also need information about the number of interactions in the path between the two persons. The `personNumFriendComments` provides the counts of comments made by friends and direct comments by the selected person. It does not however provide information about posts made by other persons, which fall in the *Message* category. This results in currently no factor table suitable to determine the amount of messages.

3.3.2 Parameter Selection

With the selected factors tables, we now need to select suitable parameters. To select the parameters from the factor table, we use two approaches depending on the factor table: a selection using window functions to select similar parameters on the entire factor table and aggregate functions to select portions of a distribution in the factor table. The selected

parameters are then stored in temporary tables and then used in the parameter selection step per query where the parameters are selected per day.

Selecting Windows

To find windows with the smallest variance in the factor table, we use window functions. The parameters are first sorted and grouped together based on the difference in frequency. Groups that are smaller than a given minimum threshold are discarded to select a group of parameters large enough to generate a sufficient amount of parameters. Finally, we select the group with the smallest standard deviation. An example of a SQL query used with DuckDB that selects personIds with a similar number of friends of friends using the described approach is shown in Listing D.1. In this example, we group the parameters where the maximum difference between the next neighbor is 5, if higher it will start a new group. Usually, the second derivative of a function gives the bending points. However, with our distribution this is not suitable given the almost linear increase of the number of friends of friends, creating windows that are too large. Therefore, using the first derivative with a threshold provides us with more control of the window sizes. Increasing the maximum difference will allow for larger groups but higher variance. Lowering the maximum difference will lower the variance and smaller group sizes, which can result in too small windows, hence a low amount of selected parameters. In Appendix C we show a visualization of the selection process. Appendix D shows an example of the selection of daily parameters for Query 1 which makes use of the resulting selected windows.

```
-- Group parameters and assign
WITH grouped_parameters AS (
    SELECT
        *, SUM(CASE WHEN Groups.diff < 5 THEN 0 ELSE 1 END)
               OVER (ORDER BY Groups.RN) AS groupId
    FROM
    (
        SELECT ROW_NUMBER() OVER(ORDER BY numFriendsOfFriends) AS RN,
               Person1Id, numFriendsOfFriends, creationDate, deletionDate,
               abs(LAG(numFriendsOfFriends, 1)
                   OVER (ORDER BY numFriendsOfFriends ASC) - numFriendsOfFriends
               ) AS diff
          FROM personNumFriendsOfFriendsOfFriends
         WHERE numFriendsOfFriends > 0
    ) Groups
),
-- Select group with smallest deviation with minimum group size
selected_group AS (
    SELECT groupId, occurrence, deviation
      FROM (
            SELECT groupId, count(groupId) AS occurrence,
```

```
22                stddev_pop(numFriendsOfFriends) as deviation
23            FROM grouped_parameters
24            GROUP BY groupId
25      ) group_stats
26      -- Miminum group size
27      WHERE group_stats.occurrence > 100
28      ORDER BY deviation ASC
29      LIMIT 1
30  )
31
32  SELECT *
33    FROM grouped_parameters, selected_group
34   WHERE grouped_parameters.groupId = (SELECT groupId FROM selected_group)
```

Listing 3.3: Example of selection of parameters using window function

Selecting Distributions

For queries where we want to select parameters that are correlated and anti-correlated, we select the parameters based on the percentile rank in the distribution using the `percentile_disc` function in DuckDB. An example is shown in Listing 3.4 where we select countries with high friendship correlation. Figure 3.12 shows the resulting selection of correlated and anti-correlated countries.

```
1  SELECT
2      country1Name AS countryXName,
3      country2Name AS countryYName,
4      frequency AS freq,
5      abs(frequency - (
6          SELECT percentile_disc(1)
7          WITHIN GROUP (ORDER BY frequency)
8            FROM countryPairsNumFriends
9      )
10     ) AS diff
11 FROM countryPairsNumFriends
12 ORDER BY diff, country1Name, country2Name
13 LIMIT 25
```

Listing 3.4: Example of selection of countries with high friendship correlation using percentiles

3.3.3 Path Curation

Interactive v2.0 has two path-finding queries: Query13 *shortest path* and a redesigned Query14, *cheapest path*. Both have two variants: one where a path is guaranteed to exist and one where no path exists. To select the paths, the factor table `people4Hops` can be used, containing the source and destination person IDs. While this factor table consists of

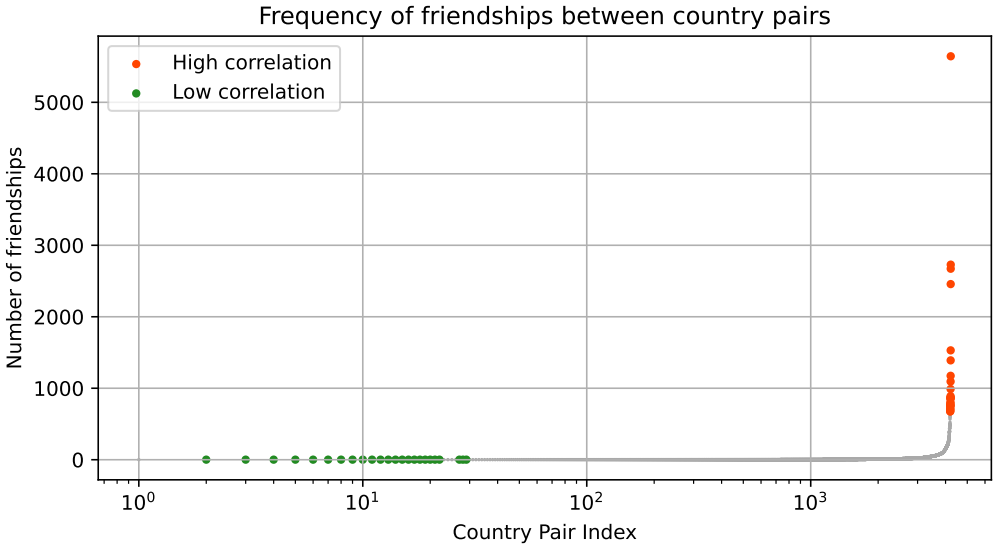

Figure 3.12: Selection of countries in the `countryPairsNumFriends` factor table on the number of friendships between both countries using the `percentile_disc` function. Correlated countries (red) are selected using percentile 1.0 and anti-correlated countries (green) are selected using percentile 0.01.

the creation date for the source and end, it does not contain information about the persons and creation dates of the edges between the nodes. The path does exist at *some* point in time, but from the table it is unknown when. Paths can exist for a temporary time interval or even multiple intervals. For v2.0, the deletion of persons and friendship edges in the path also needs to be considered. An example of this problem is given in Figure 3.13, with three hops for simplicity. While the problem of the statistics changing every day in other factor tables, this project focuses on solving the problems with the paths, since not curating the paths makes it difficult to distinguish the behaviors for the introduced variants.

To select valid paths, a table containing all valid paths with four hops distance is created before generating the parameters. This is done by loading the person nodes with their friendship edges into a network modeling library, `networkit` (57), where the benchmark is played per simulation day of the benchmark. Since it is not feasible to compute all shortest paths in the graph for larger scale factors, only the top 10 person IDs close to the median of the number of `numFriendsOfFriendsOfFriends` are taken into consideration. The algorithm for the path selection is shown in Algorithm 1. The path from the source person to all other persons in the graph is then computed and only the 4-hop paths are stored, together with the day it is valid. Smaller or larger hop paths are discarded.

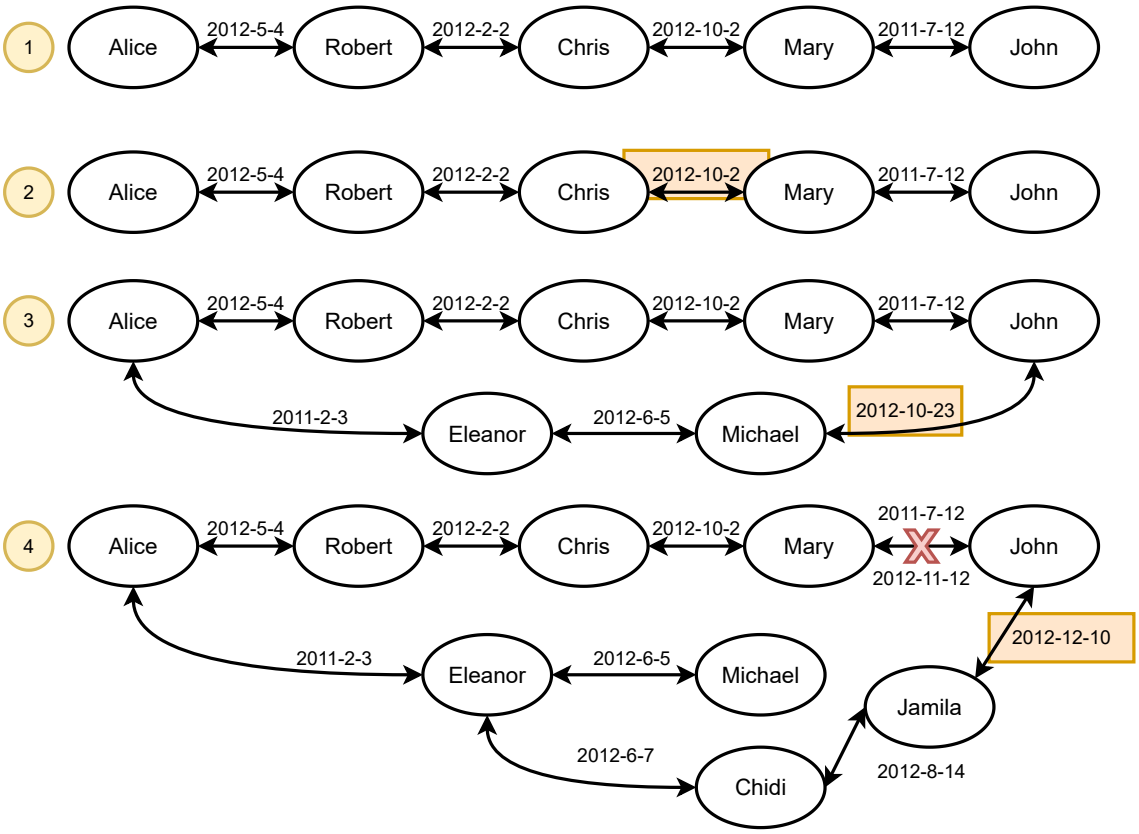

Figure 3.13: Example of the problem of temporary paths in the Spark Data set.
(1) We start with a 3-hop path between Alice and John with the friendship edges labeled with their creation date.
(2) Suppose the benchmark starts at `2012-9-1`, the path between Chris and Mary does not exist until `2012-10-2`, making the path not suitable for the path queries where a path should be guaranteed.
(3) The path can be shortened if friendship edges are added, e.g., between Michael and John on `2012-10-23`, reducing the total hops, thus lowering the complexity of the path between Alice and John.
(4) Another possibility is that the edge between Mary and John is deleted at `2012-11-12`, but another path is formed at a later date, `2012-12-10`, making the path valid again.

Algorithm 1 Path Curation algorithm for each day

Input: G, T_{start}, T_{end}, $N_{sources}$, $N_{targets}$
Output: node pairs with 4 hops during T_{start}, T_{end}
  for N in $N_{sources}$ **do**
    $M_{shortest\,paths\,distances}$ = MultiTargetBFS(G, N,$N_{targets}$)
    select *IDs* from M where distance == 4
    store *IDs* together T_{start}, T_{end}
  end for

3.4 Updating the Driver

With the new update streams stored in a Parquet file grouped by the update query and substitution parameters with a validity period, the driver needs to be updated to accommodate these changes. The interactive driver, implemented in Java, previously determined the number of write threads by the number of files exported by the Hadoop Datagen and the update streams were grouped by person update or forum and friendship updates, assigning one file per thread. This is not feasible with the new update streams since these are grouped by query and the number of updates is different per query.

Scalable Update Stream Loading

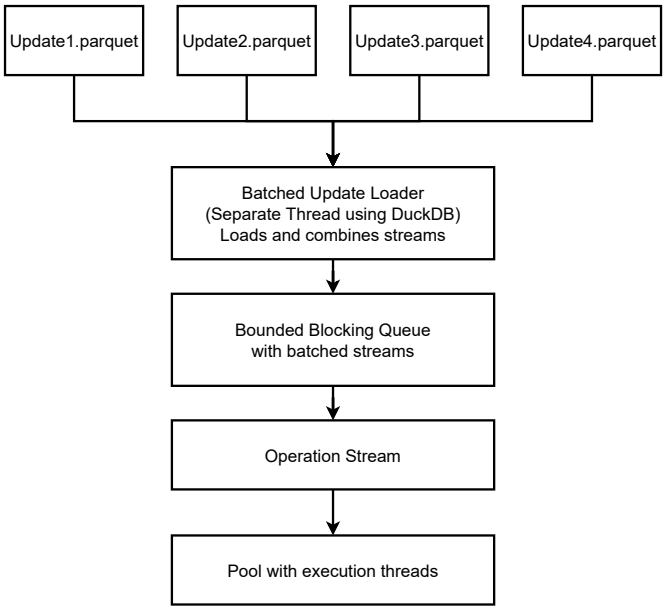

Figure 3.14: Batched loading of the update streams using DuckDB

In v2.0, the way the update streams are loaded is changed: instead of the update streams being executed in separate write threads, the update streams are executed within the same threadpool as the reads, requiring only one threadpool, reducing synchronization overhead between the separate threads when updating the completion time. In addition, using one threadpool reduces the idle threads: in v1.0 the write threads, especially the ones only taking person updates into account, were mostly idle. The total threads used by the driver is therefore the amount specified by the user using the properties file with the `thread_count` property. An additional benefit of this approach is that the data does not have to be

generated for different numbers of write threads.

To enable the update stream queries to be executed in the same threadpool of the read queries, each update stream is loaded by creating a view on the Parquet file using DuckDB[1], running in the Java Virtual Machine (JVM). The data is then batch loaded using a separate thread, where from each update stream the data is loaded for a given *batch interval*, which is the amount of hours in simulation time to load. The batch interval is the simulation time window to load from the updates, reducing the memory usage of the driver when using larger scale factors. The loaded data is then merged into one stream sorted on the scheduled time of the updates. The stream is added to a blocking *first-in-first-out* (FIFO) queue where the driver fetches the stream. The blocking queue is bounded and when the queue is full, the loader will block until a slot is open. This way, when the driver requests a new stream, there is always one available to prevent delay during the benchmark execution. When the benchmark execution stops, the thread is interrupted and stopped gracefully. The workflow is shown in Figure 3.14.

To set a suitable batch size, the mean updates per hour per scale factor needs to be taken into account together with the throughput of the SUT. The batch size needs to be large enough such that the batch loader loading time is smaller compared to the time it takes to execute one batch.

Complex query execution using temporal parameters The substitution parameters are loaded in the driver once and added to a generator where each complex query read is generated according to the specified intervals. However, the execution of the complex queries is changed in v2.0: the driver only executes a query if the parameter is in the scheduled time interval. Therefore, each complex read operation has two properties: the dependency time, the time from which it is valid to execute the query with given parameters and an expiry date: the time the parameter cannot be used, for example, because a dependency has been deleted. When the driver encounters a query that is not in the valid time window, it skips to the next query.

[1] `https://duckdb.org/docs/data/parquet#inserts-and-views`

3.5 Updating the Reference Implementations

The LDBC SNB Interactive workload has reference implementations to show an example of how to correctly implement the workload. For Interactive v2.0, we updated the reference implementations for Neo4j, Postgres and Umbra [1] to include delete operations and support for the data set produced by the Spark Datagen. To support the cascading delete operations, Postgres and Umbra use foreign key constraints with `ON DELETE CASCADE` to respect the semantic constraints. When, e.g., a person is deleted, the cascading delete will remove linked forums, posts, descendant comments as well, as well as its edges. This ensures that there are no dangling edges in the graph. With Neo4j, using `OPTIONAL MATCH` in the delete query ensures cascading deletes to, e.g., replies of a comment that is deleted. For both the Umbra and Postgres implementations, we provided a loader that can be started in a Docker container to remove the need to install dependencies, improving the usability of the framework. Using the Docker Compose Tool, `docker-compose`[2], the loader is started alongside the DBMS instance in a container with its dependencies and starts loading automatically once the database has started.

3.6 SQL Server Reference Implementation

For LDBC SNB Interactive v2.0, SQL Server is added as an additional reference implementation. This implementation makes use of Transact-SQL (T-SQL) including the SQL Graph capabilities (see Section 2.4.5). As a basis, the queries from the Postgres implementation, adopting the SQL:1999 queries written in PostgreSQL dialect to T-SQL. The creation of this new reference implementation required writing several components from scratch:

- Batch data loader: the loader for SQL Server requires to specify for each table a format file to support the correct loading of the datetime values and Unicode (see Section 2.6).
- SQL Graph requires a changed schema and bulk loader to define the table where the node and edge information are stored.
- To mitigate errors when installing the dependencies of the SQL Server driver[3] for the loader, the loader is containerized to isolate the dependencies using `docker-compose`.

[1] Umbra's queries, loader and schema implementation were upgraded by Gabor Szarnyas (CWI) and Altan Birler (TUM).

[2] `https://docs.docker.com/compose/`

[3] `https://learn.microsoft.com/en-us/sql/connect/odbc/linux-mac/installing-the-microsoft-odbc-driver-for-sql-server`

- For Query 14, a single-source BFS algorithm is written in T-SQL and saved as a stored procedure. The weights are precomputed and maintained during inserts and deletes.

Of the rewritten queries, complex queries 2, 12 and 13, and short queries 2, 3 and 6 leverage SQL Graph capabilities using the `MATCH` and `SHORTEST_PATH` functions. The `SHORTEST_PATH` function finds unweighted shortest paths between the specified nodes. Additionally, when inserting an edge, SQL Graph requires additional syntax to retrieve the node information, summarized in the `NODE_ID`[1] property, to form an edge.

To implement cascading deletes with SQL Server using the SQL Graph syntax, we make use of triggers: this is due to a limitation in SQL Server that a table with self-referencing foreign keys, e.g., the table with posts and comments, creates multiple cascade paths once a delete is executed[2].

[1] https://learn.microsoft.com/en-us/sql/t-sql/functions/node-id-from-parts-transact-sql

[2] https://learn.microsoft.com/en-US/sql/relational-databases/errors-events/mssqlserver-1785-database-engine-error?view=sql-server-ver16

4

Evaluation of Interactive v2.0

4.1 Experimental Setup

We executed several experiments to evaluate the parameter generator and curation (explained in Section 3.3), as well as evaluating the runtimes of delete queries. For these experiments, we used the following setup:

- Azure Compute `Standard_E16d_v5`, 16 vCPU, Intel(R) Xeon(R) 8370 @ 2.80 GHz
- 128 GB RAM, 600 GB SSD
- Operating system: Ubuntu 22.04 LTS (kernel version: 5.15.0-1022-azure)
- Java version: 11.0.17+8-post-Ubuntu-1ubuntu222.04
- Docker version: 20.10.21
- File system: ext4
- Python version: 3.10
- DuckDB version (used in Paramgen): 0.5.1

For benchmarks to compare the total runtimes of the complex reads between Interactive v1.0 and v2.0, we used:

- Azure Compute `Standard_L16s_v3`, 16 vCPU, Intel(R) Xeon(R) 8370 @ 2.80 GHz
- 128 GB RAM, 3.4 TB NVMe SSD (2×1.7 TB in RAID0)
- Operating system: Ubuntu 22.04 LTS (kernel version: 5.15.0-1022-azure)
- Java version: 11.0.17+8-post-Ubuntu-1ubuntu222.04
- Docker version: 20.10.21
- File system: ext4
- Python version: 3.10

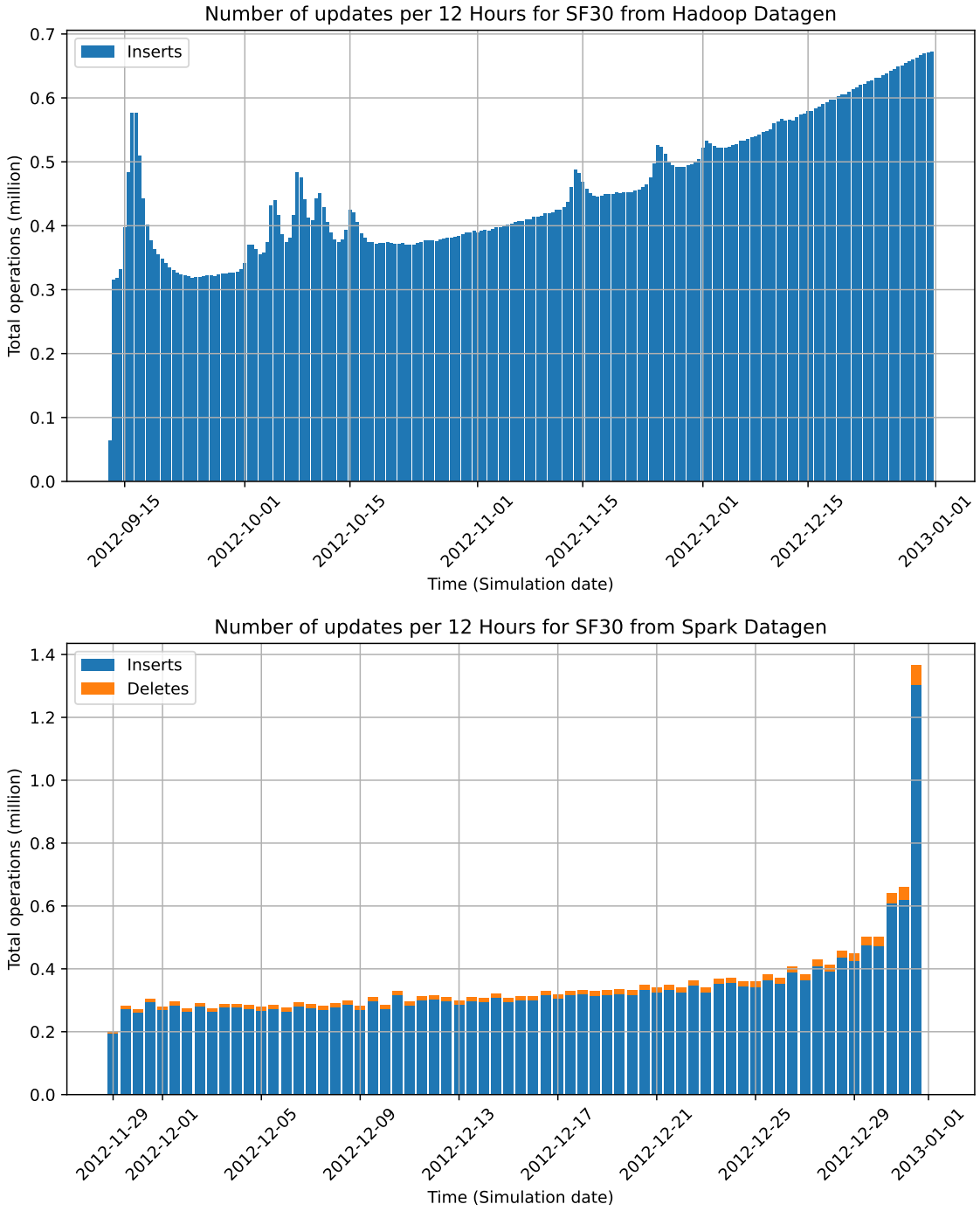

Figure 4.1: Number of events per 12 hours for Hadoop and Spark during the simulation time of the benchmark.

To compare the runtimes of the parameter generation scripts between Interactive v1.0 and v2.0, we used:

- Intel(R) Xeon(R) E5-4657L v2 @ 2.40 GHz, 48 cores with Hyper-Threading
- 1 TiB RAM, HDD
- Operating System: Fedora 36 (kernel version: 5.18.18-200.fc36.x86_64)
- Python version: 3.10
- DuckDB version: 0.5.1

To reproduce the test environments in Azure, scripts are used to provision and configure each virtual machine. For provisioning of the used infrastructure, Terraform[1] is used and for configuration of the VM, Ansible[2] is used. These scripts are provided in a public repository[3].

Database systems The experiments to compare the runtimes with v1.0 and v2.0 were executed against Neo4j Enterprise Edition `4.4.1`[4] and Umbra `45f3aae27`, both running in a Docker container. Additionally, we use an RDBMS, anonymously referred to as *DBMS X* to demonstrate the benchmark's portability and gain an insight into the expected performance of RDBMSs.

4.2 Experiments for Tuning SNB Interactive v2

4.2.1 Characterization of the Hadoop and Spark Datagen's Data Sets

The migration to the Spark data set (Section 2.6.1) introduces the delete events in the Interactive workload. We observed multiple differences between the Hadoop and Spark data sets affecting the Interactive workload, in addition to the differences explained in Section 2.6.1. The number of updates for a given time differs: Figure 4.1 shows the number of updates per 12 hours of simulation time for scale factor 30. The Hadoop Datagen has on average more updates and exhibits spiking events, while the Spark Datagen mostly shows a linear increase without spiking events until the last 72 hours where the number of updates shows an exponential increase. This is not a desirable distribution since if a SUT with a high TCR^{-1} is fast enough to reach the last 12 hours, it can slow down due to the amount of update queries in the last period. In addition, the total number of updates is lower in

[1] `https://www.terraform.io`
[2] `https://www.ansible.com`
[3] `https://github.com/GLaDAP/cloud-bootstrap`
[4] `https://hub.docker.com/_/neo4j`

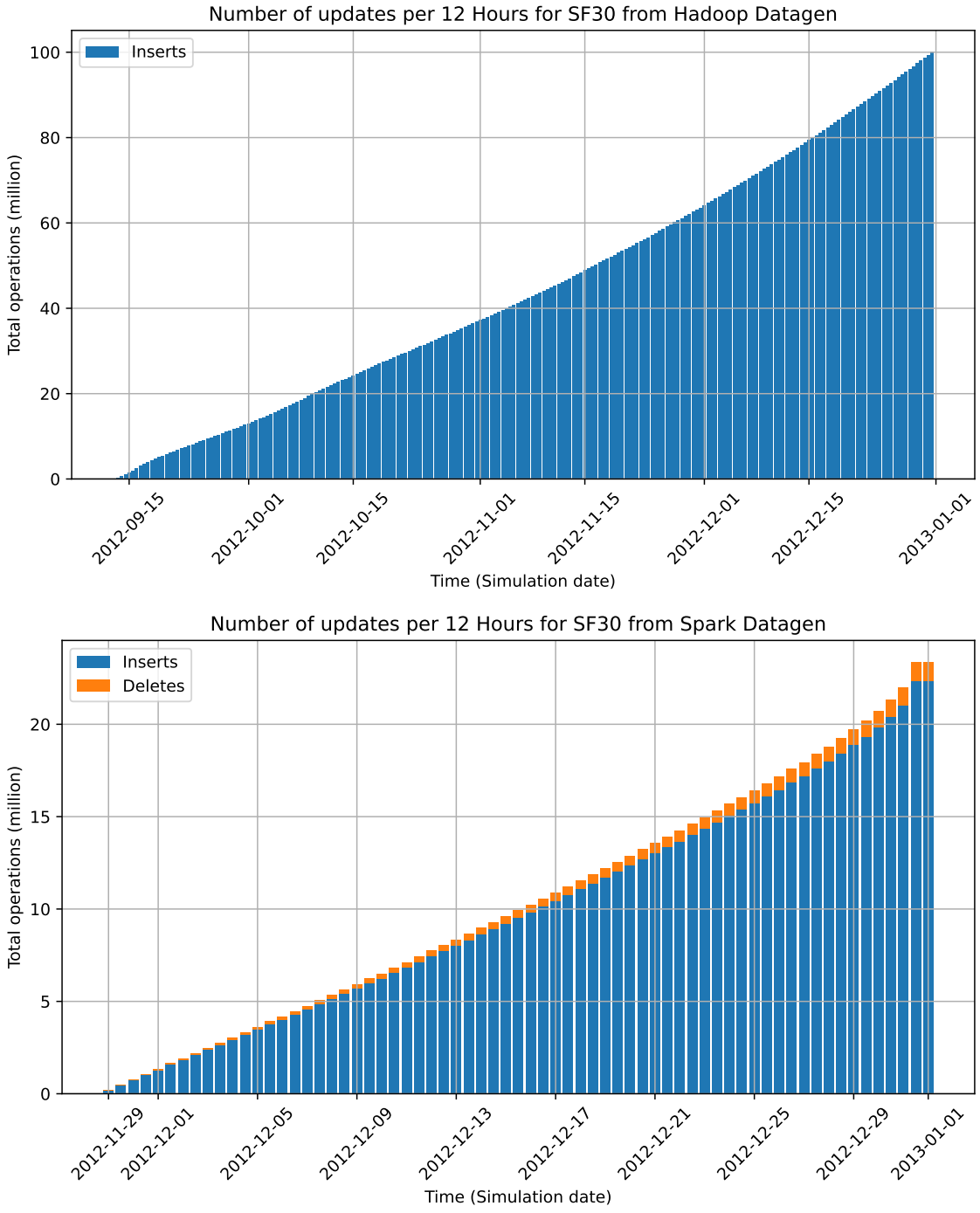

Figure 4.2: Cumulative sum of the number of events per 12 hours for Hadoop and Spark during the simulation time of the benchmark.

the data set generated by Spark compared to the Hadoop one.

Figure 4.2 shows the number of updates executed per 12 simulation hours. The timespan of the data set produced by Spark also affects the upper bound of the TCR^{-1} value. For a valid benchmark run, the warmup and measurement window (see Section 2.6.5.3) should be a minimum of 2.5 hours in total. Spark Datagen leaves 3% of the simulation time for the update streams, resulting in $3 \times 365 \times 0.03 \times 24 = 788.4$ hours for the update streams, resulting in a lower bound TCR^{-1} of $\frac{2.5}{788.4} = 0.0034$. This is higher than the lower bound TCR^{-1} when using the data set from Hadoop, 0.001 (1, Section 7.4.7.2).

4.2.2 Parameter Curation

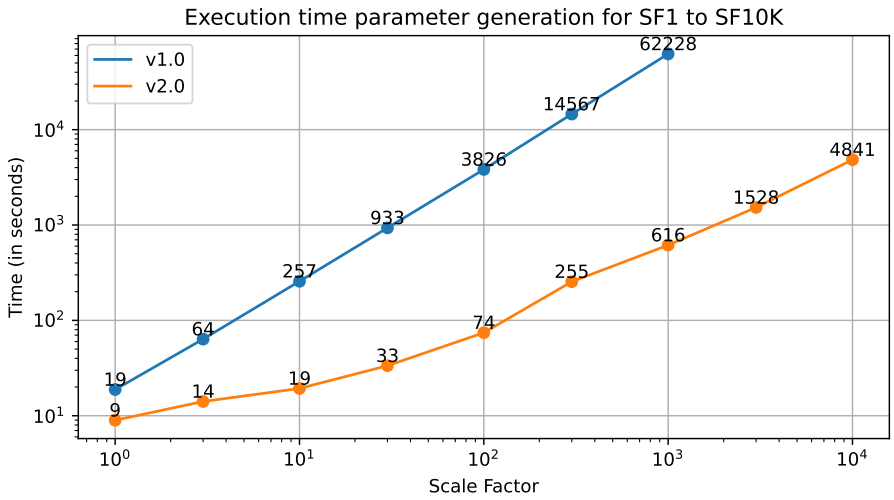

Figure 4.3: Runtime of the parameter generators for Interactive v1.0 and v2.0. The labels show the runtime in seconds.

Scaling Parameter Generation

To improve the scalability of the parameter generator, we wrote a new parameter generator to support larger scale factors with support for temporal parameters. Figure 4.3 shows the runtimes of the parameter generators for Interactive v1.0 and v2.0. As can be seen, the new parameter generator scales well for larger scale factors. The new parameter generator is more than two orders of magnitude faster on SF1,000 and is able to scale to SF10,000 while only using a maximum of 80 GB of memory.

	v1.0	v2.0
Threads	1 read and 2 write threads	1
Instance	Azure Compute Standard_L16s_v3	
Warmup	30,000	
Operation count	120,000	

Table 4.1: Benchmark environment used in parameter curation comparison

Comparison of Parameter Generation v1.0 and v2.0

When comparing the runtimes and curated parameters from Interactive v1.0 with Interactive v2.0, we used the configuration shown in Table 4.1. We evaluated queries 1 to 12 on SF30 for Neo4j and Umbra.

Figure 4.4 and Table 4.2 show the runtime distribution and statistics using Neo4j. When looking at the standard deviation of the runtimes of the queries when using Neo4j, we observe that the parameter generation in Interactive v2.0 is only better performing for queries 1, 5, 8 and 9 compared to v1.0. Figure 4.5 and Table 4.2 show the results using Umbra. When executed with Umbra, queries 1–5 and 10 exhibit a lower standard deviation using the parameters generated by Interactive v2.0 compared to v1.0. The runtimes for queries 7 and 8 are significantly higher in v2.0 compared to v1.0. This is because, in v2.0, the selected Person IDs have more friendship connections, therefore a higher chance of interactions in the Post subgraph. The factor tables in the Spark Datagen contain the statistics for the entire simulation time, and therefore do not take temporal changes into account. This leads to inaccuracies in the estimated intermediate cardinalities, affecting parameter curation. We provide an example of this in Section 4.2.3.

For the newly introduced variants for Query 3, correlated and anti-correlated countries, the percentile selected for country pairs does not affect the runtimes: correlated and anti-correlated countries show similar runtime behavior. This could indicate that the two cases do not exhibit a significant difference and/or the DBMSs used in this experiment picked the correct query plans for these queries.

4.2.3 Path Curation

For path curation, we loaded the person and friendship graph into `networkit` (57) and batch load the person and friendship updates per day (see Section 3.3.3). The parameter curation relies on the number of 1-, 2- and 3-hop friends for a given person. Unfortunately,

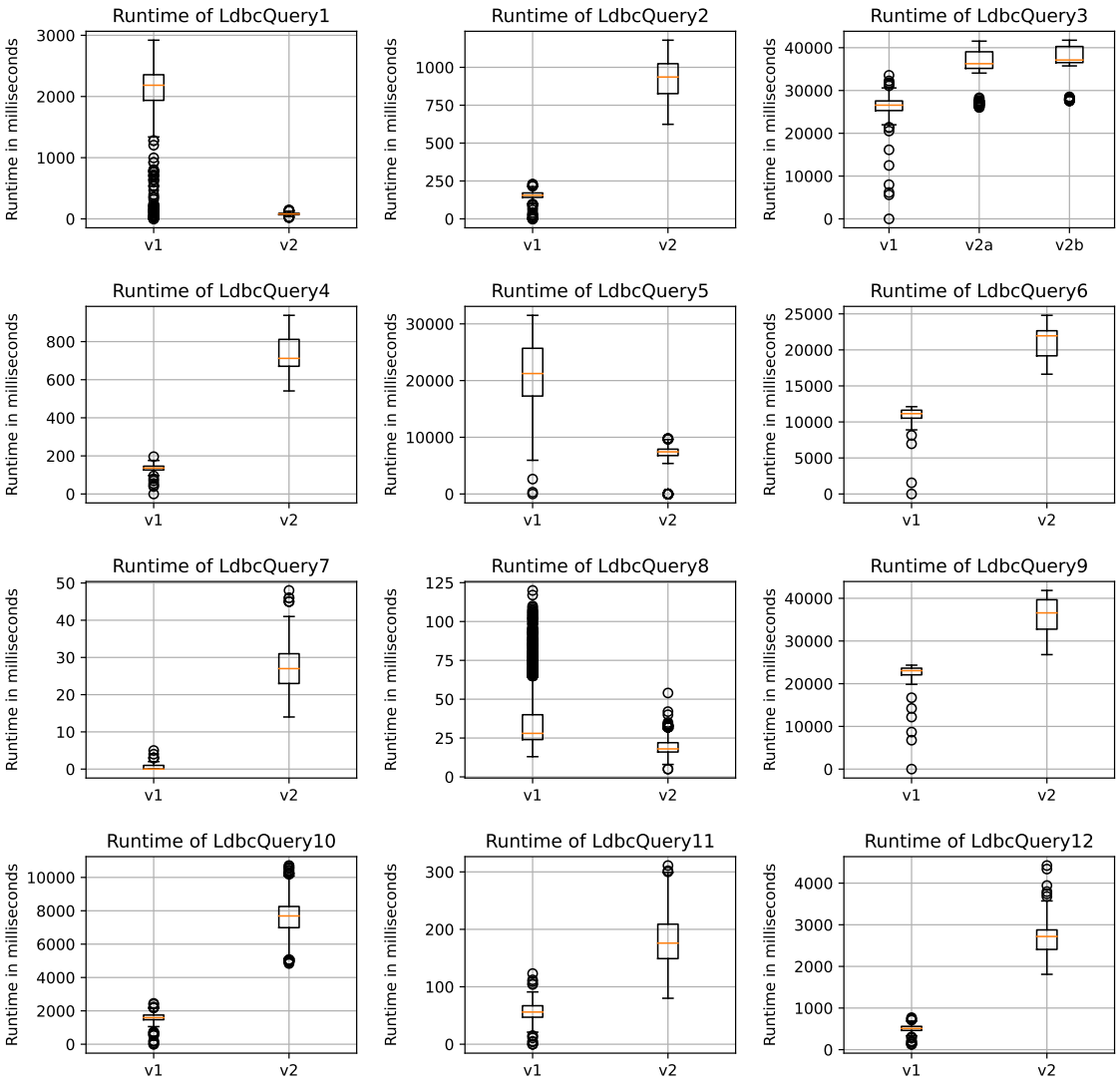

Figure 4.4: Parameter Curation: Curated v1.0 vs. curated v2.0 using Neo4j with SF30. For query3, v2a and v2b denote the variants of the query.

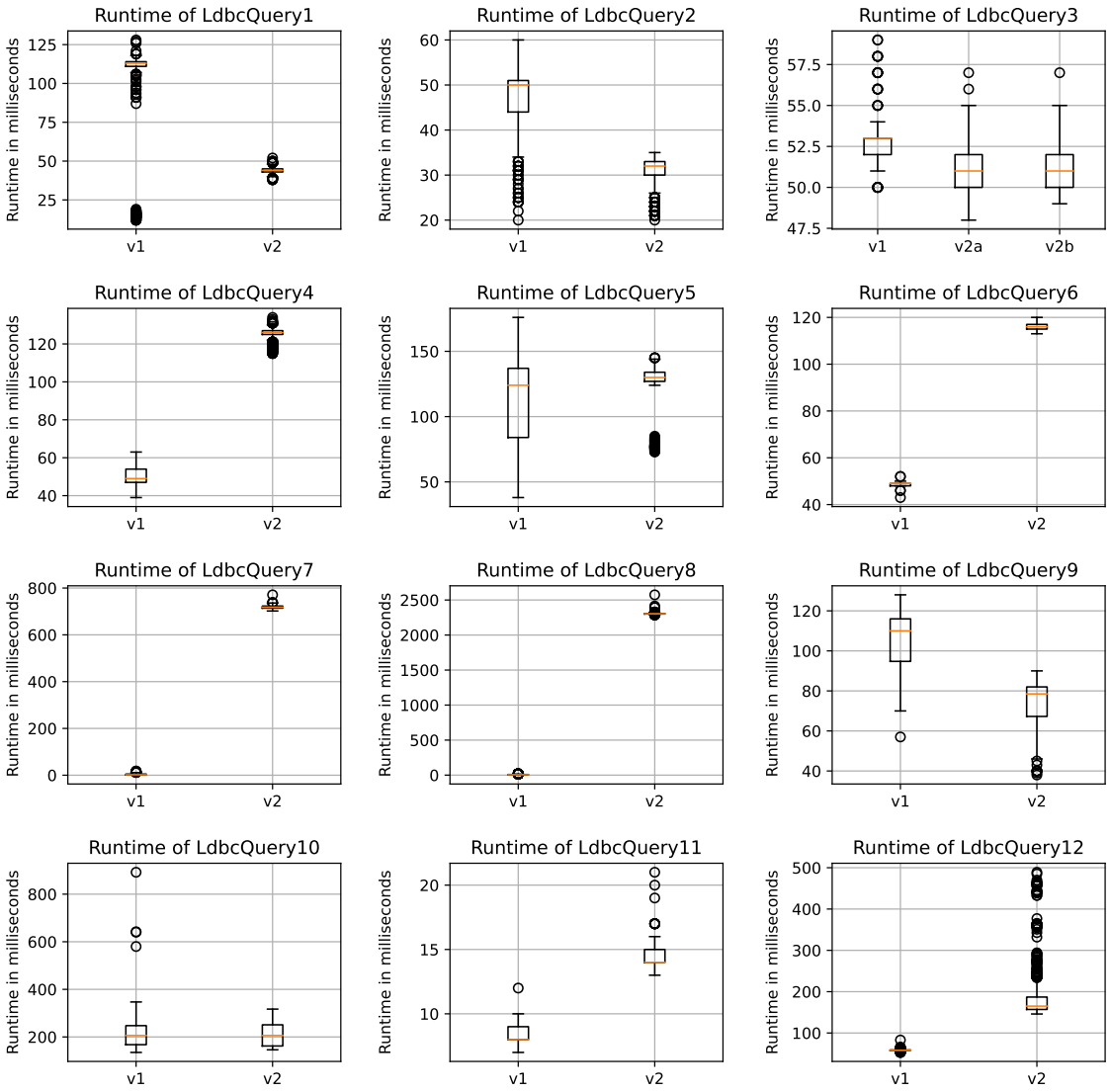

Figure 4.5: Parameter Curation: Curated v1.0 vs. curated v2.0 using Umbra `45f3aae27` with SF30. For query3, v2a and v2b denote the variants of the query.

Version	Query	Neo4j standard deviation	Neo4j mean runtime	Umbra standard deviation	Umbra mean runtime
v1	LdbcQuery1	640.17	2,006.48	23.30	106.63
v2	LdbcQuery1	17.86	80.16	2.03	44.11
v1	LdbcQuery2	29.94	153.43	6.95	47.30
v2	LdbcQuery2	134.20	921.21	3.79	30.22
v1	LdbcQuery3	3,735.94	26,001.75	1.62	52.74
v2a	LdbcQuery3a	4,482.94	35,454.43	1.67	51.39
v2b	LdbcQuery3b	4,094.59	37,043.01	1.55	51.18
v1	LdbcQuery4	17.55	135.57	4.63	50.04
v2	LdbcQuery4	97.88	724.82	3.69	125.21
v1	LdbcQuery5	5,480.80	21,240.52	35.97	109.85
v2	LdbcQuery5	2,633.08	6,648.29	18.39	125.21
v1	LdbcQuery6	1,935.23	10,649.56	1.05	48.52
v2	LdbcQuery6	2,388.11	21,276.04	1.37	116.12
v1	LdbcQuery7	0.61	0.31	3.11	3.50
v2	LdbcQuery7	5.88	26.80	6.65	718.36
v1	LdbcQuery8	17.42	35.12	3.29	7.49
v2	LdbcQuery8	4.42	18.72	7.67	2,304.18
v1	LdbcQuery9	4,465.42	21,596.71	15.46	104.91
v2	LdbcQuery9	4,300.84	36,095.18	15.50	71.95
v1	LdbcQuery10	279.87	1,594.01	62.65	211.87
v2	LdbcQuery10	1,092.73	7,695.16	51.22	213.98
v1	LdbcQuery11	14.21	56.58	0.52	8.31
v2	LdbcQuery11	40.56	180.05	0.84	14.25
v1	LdbcQuery12	80.30	508.63	1.78	58.35
v2	LdbcQuery12	361.59	2,650.06	64.81	189.69

Table 4.2: Comparison of variance, standard deviation and mean of runtimes in milliseconds using Neo4j and Umbra on SF30.

Occurrences in time	Number
One occurrence	142,696
Two occurrences	3,578
Three occurrences	88

Table 4.3: Total number of paths with discontinuous time intervals for SF10.

the factor generator produces overapproximated results for these values. Figure 4.6 shows the effect of the changes in the friendships for the *number of friends of friends* for a given person in SF10. The person's *number of friends of friends* defined in the factor table is never reached during the simulation time of the social network, with the accuracy varying between 62% and 86%. This inaccuracy in the factor table affects the parameter curation for the queries that use the factor table with the 1-, 2- and 3-hop friends. We also see the effect of deletes, where at `2012-12-27`, a friendship is deleted, reducing the total of friend of friends.

Besides the number of friends of friends, updates in the graph also affect the 4-hop paths. Table 4.3 shows the number of 4-hop paths found in the Spark SF10 data set with

discontinuous time intervals. Without information about when a path is 4 hops long, the same parameter will give different runtimes since the path has a different hop count, often smaller than 4 hops.

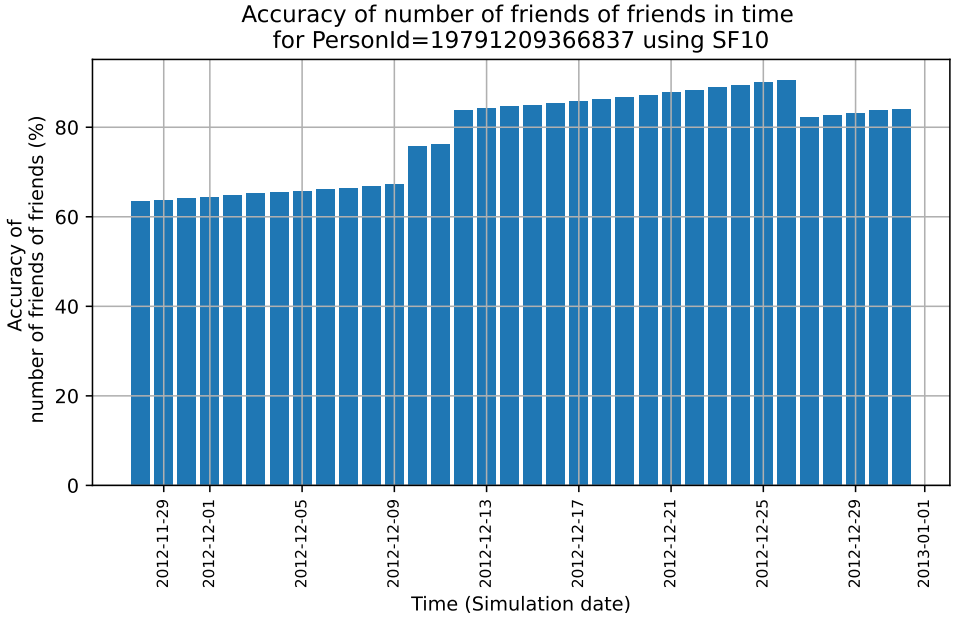

Figure 4.6: Example of *number of friends of friends* for a given person ID in the SF10 data set. The accuracy of the *number of friends of friends* varies for one person per day during the simulation timeframe between 62% and 86%.

Path Queries

Interactive v2.0 introduces variants to the path queries and a new definition for Query 14 to find the cheapest path between two persons. The query template of Query 13, finding an unweighted shortest path between two nodes, is unchanged. In Interactive v2.0, both queries have two variants that differ in their input parameters: one variant guarantees the absence of a path, and the other guarantees the existence of a path. Figure 4.7 shows the runtimes of Query 13 and Query 14 using v1.0 and v2.0. For Query 13, the runtimes are comparable to v1.0. When a path is guaranteed, we see on average longer runtimes compared to the no-path variant.

For Query 14, we observe similar runtimes for the variants. This is due to Neo4j's limitation of using a unidirectional Dijkstra path-finding algorithm in their *Graph Data Science* (GDS) library.[1] However, while Query 14's specifications are different between

[1] `https://neo4j.com/docs/graph-data-science/current/algorithms/dijkstra-source-target/`

v1.0 and v2.0, Query 14 for v2.0 has significantly higher runtimes compared to v1.0. In v1.0, Query 14 requires all shortest paths between two persons and then calculates the weights of each path, while in v2.0 the query requires the cheapest path.

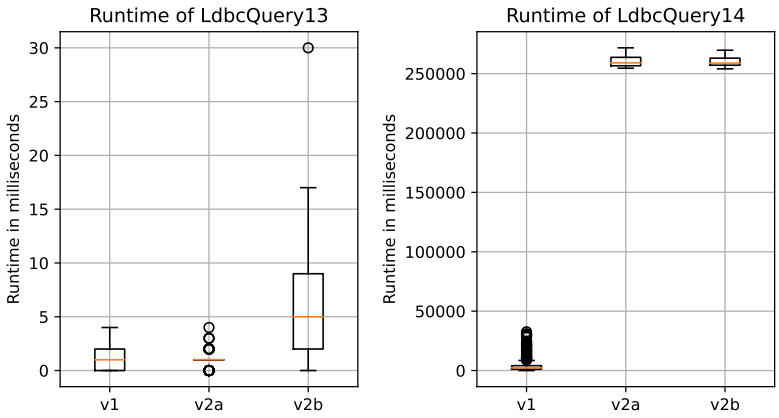

Figure 4.7: Path queries: Runtimes for path queries with interactive v1.0 and interactive v2.0 using Neo4j. Note that Q14 is changed between v1.0 and v2.0, with v2.0 performing the computationally much more difficult cheapest path problem.

4.3 Effect of Deletes

To measure the effect of deletes, we used Neo4j Enterprise Edition 4.4.1 using 1 thread with only the insert and delete queries enabled. Each run consists of 1,000,000 update operations to execute a sufficient number of delete operations from the update streams to allow studying their effect.

Figure 4.8 shows the runtimes of the delete queries for scale factors 10, 30, 100, and 100, while Table 4.4 shows statistics for each delete query, including the number of times the query is executed. On average, removing a like to a post/comment or a forum membership, takes the same amount of time for all scale factors, indicating that the delete performance of Neo4j scales well when increasing the size of the graph. This is likely due to the storage design in Neo4j where a linked list of pointers is used to refer to the nodes and edges (8). However, for deleting a forum, subthread or a person from the network, the average and maximum runtimes increase, but not significantly. When the scale factor increases, the runtime for deleting a person increases the most since removing a person also removes all their edges, comments and forums, causing multiple implicit delete operations after explicitly deleting a person.

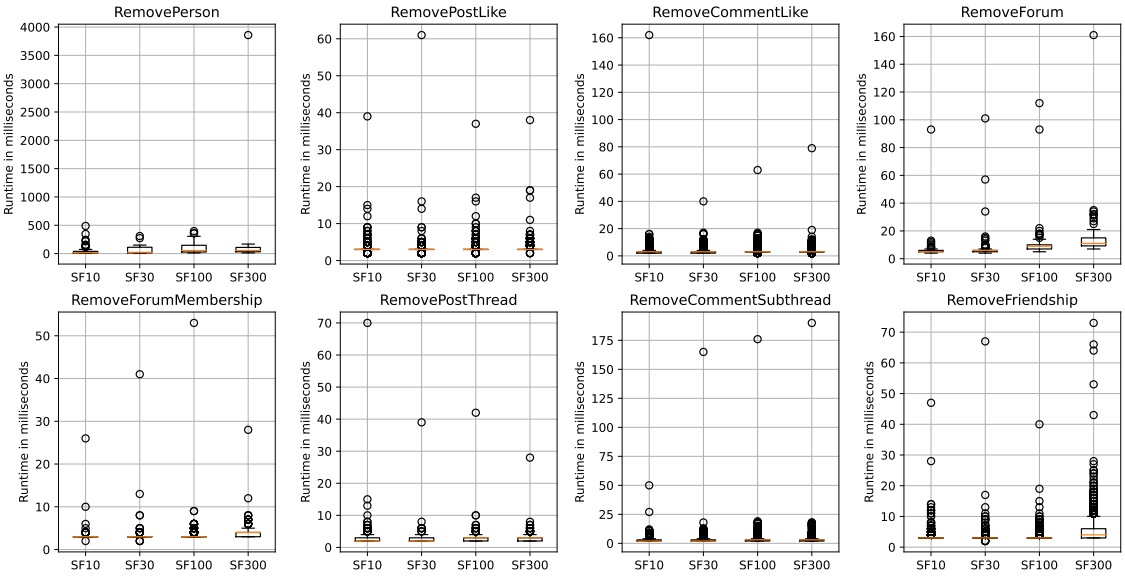

Figure 4.8: Effect of deletes: Runtimes of delete queries using Neo4j 4.4.1 for scale factors SF10, 30, 100 and 300

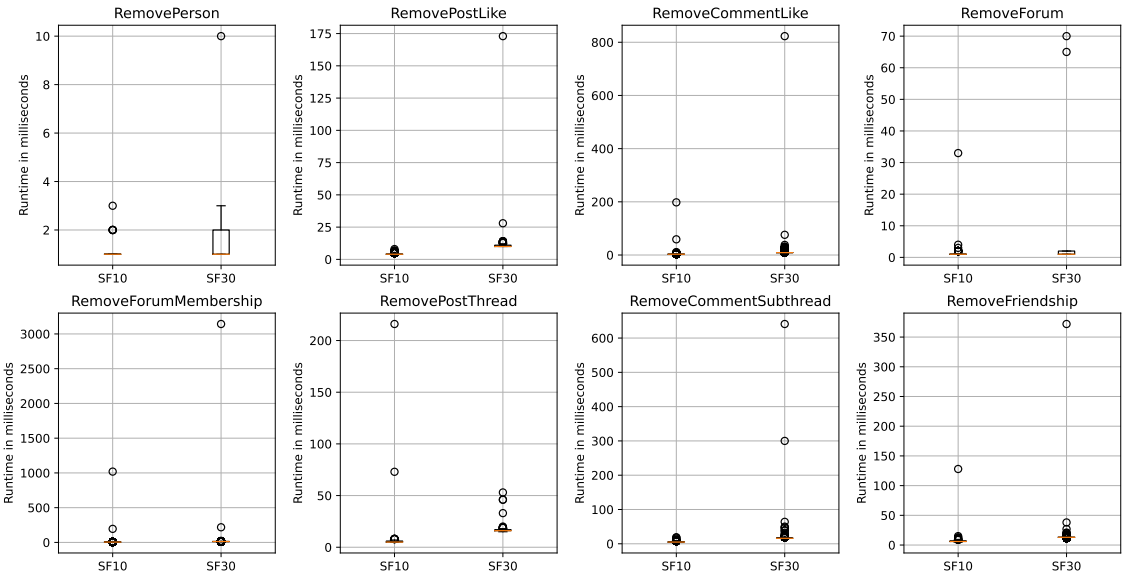

Figure 4.9: Effect of deletes: Runtimes of delete queries using Umbra for scale factors SF10 and SF30.

Query	SF	Count	Min	Max	Mean	P5	P95
LdbcDelete1RemovePerson	10	29	8.00	347.00	54.79	9.00	209.20
	30	19	10.00	310.00	82.63	11.80	276.70
	100	21	12.00	397.00	111.62	19.00	361.00
	300	19	13.00	3,858.00	267.68	22.90	537.00
LdbcDelete2RemovePostLike	10	2430	2.00	39.00	2.93	2.00	3.00
	30	2495	2.00	61.00	3.02	2.00	3.00
	100	2541	2.00	37.00	3.12	3.00	4.00
	300	2824	2.00	38.00	3.26	3.00	4.00
LdbcDelete3RemoveCommentLike	10	14660	2.00	162.00	2.64	2.00	3.00
	30	15256	2.00	40.00	2.70	2.00	3.00
	100	14538	2.00	63.00	3.11	3.00	4.00
	300	16696	2.00	79.00	3.30	3.00	4.00
LdbcDelete4RemoveForum	10	136	4.00	93.00	6.10	4.00	8.00
	30	138	4.00	101.00	7.27	4.00	10.45
	100	109	5.00	112.00	10.94	6.00	17.60
	300	110	7.00	161.00	14.79	9.00	30.10
LdbcDelete5RemoveForumMembership	10	859	3.00	26.00	3.06	3.00	3.00
	30	669	2.00	41.00	3.14	3.00	4.00
	100	569	3.00	53.00	3.32	3.00	4.00
	300	543	3.00	28.00	3.78	3.00	5.00
LdbcDelete6RemovePostThread	10	2130	2.00	70.00	2.56	2.00	3.00
	30	1710	2.00	39.00	2.50	2.00	3.00
	100	1627	2.00	42.00	2.62	2.00	3.00
	300	1523	2.00	28.00	2.64	2.00	3.00
LdbcDelete7RemoveCommentSubthread	10	12967	2.00	50.00	2.54	2.00	3.00
	30	12200	2.00	165.00	2.55	2.00	3.00
	100	12133	2.00	176.00	2.66	2.00	3.00
	300	11837	2.00	190.00	2.66	2.00	3.00
LdbcDelete8RemoveFriendship	10	4812	3.00	47.00	3.09	3.00	3.00
	30	4900	2.00	67.00	3.08	3.00	3.00
	100	5097	3.00	40.00	3.15	3.00	4.00
	300	5153	3.00	73.00	4.97	3.00	10.00

Table 4.4: Runtimes for deletes using Neo4j v4.4.1 for SF10, 30, 100 and 300.

Query	SF	Count	Min	Max	Mean	P5	P95
LdbcDelete1RemovePerson	10	29	1.00	3.00	1.24	1.00	2.00
	30	19	1.00	10.00	1.79	1.00	3.70
LdbcDelete2RemovePostLike	10	2430	4.00	8.00	4.17	4.00	5.00
	30	2495	10.00	173.00	10.48	10.00	11.00
LdbcDelete3RemoveCommentLike	10	14660	3.00	198.00	3.11	3.00	4.00
	30	15256	9.00	823.00	9.29	9.00	10.00
LdbcDelete4RemoveForum	10	136	1.00	33.00	1.49	1.00	2.00
	30	138	1.00	70.00	2.28	1.00	2.00
LdbcDelete5RemoveForumMembership	10	859	5.00	1,019.00	6.61	5.00	6.00
	30	669	12.00	3,143.00	17.90	12.00	14.00
LdbcDelete6RemovePostThread	10	2130	5.00	216.00	5.53	5.00	7.00
	30	1710	15.00	53.00	16.23	15.00	18.00
LdbcDelete7RemoveCommentSubthread	10	12967	5.00	19.00	5.49	5.00	7.00
	30	12200	15.00	641.00	16.34	15.00	18.00
LdbcDelete8RemoveFriendship	10	4812	5.00	128.00	6.38	6.00	7.00
	30	4900	12.00	372.00	13.11	12.00	14.00

Table 4.5: Runtimes for deletes using Umbra for SF10 and 30.

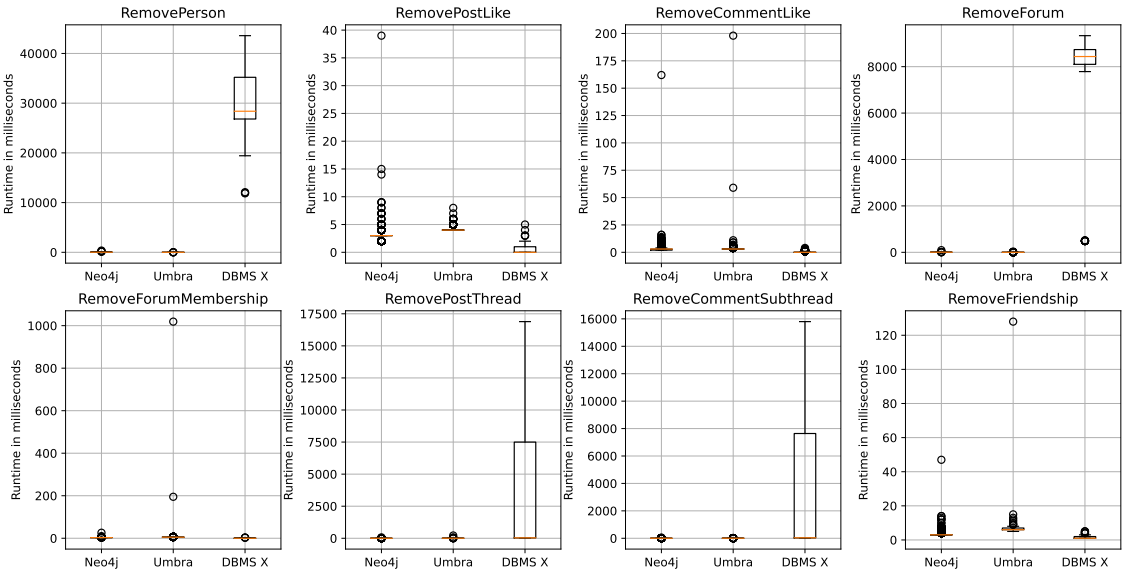

Figure 4.10: Effect of deletes: Runtime comparison of delete queries for Neo4j v4.4.1, Umbra and DBMS X using SF10 and 750,000 update operations.

Figure 4.9 shows the runtimes of the delete queries for Umbra on SF10 and SF30 and Table 4.5 show the statistics. We were unable to obtain results for SF100 and larger since Umbra failed due to memory exhaustion. The results show that the mean runtimes of the deletes increase when increasing the scale factor. Figure 4.10 shows the runtimes for delete queries for Neo4j, Umbra and DBMS X using SF10. The runtime for deleting a node, with cascading deletes, takes significantly more time with DBMS X than with Neo4j. Deletion of edges on the other hand is faster using DBMS X.

With Interactive v2.0, the delete operations can result in a deadlock on the SUT when the TCR^{-1} is set too high and the benchmark is running multithreaded. This results in the system becoming unable to keep up with the delete operations. For example, DBMS X runs into a deadlock and aborts one of the deadlocked transactions, which in turn results in the benchmark being aborted. This is because the transactions with delete operations, especially when there are multiple cascading delete operations, will lock resources to which the DBMS ultimately decides to rollback one of the deadlocked transactions.

5

Related Work on Database Benchmarks

This section provides information about other benchmarks that target systems capable of running graph workloads, benchmarks targetting OLTP DBMSs, and information about current work on distributed benchmark frameworks.

5.1 LDBC SNB Business Intelligence (BI)

The LDBC SNB BI workload tests DBMSs capable of OLAP workloads on a graph analytics workload, using the data generated by the LDBC SNB Spark Datagen. (60) The queries contain complex aggregations and joins that touch large parts of the data. In contrast with SNB Interactive, BI explores large portions of the social network graph to search for occurrences of graph patterns, causing large intermediate result sets which are challenging to systems that do not prune the search space efficiently.(61) The benchmark contains 20 complex read queries, 8 insert queries, which are the same as the ones in SNB-I, and 8 delete queries. (1) It uses parameter curation and defines variants for 8 queries, leading to a total of 28 query variants. SNB-BI defines two performance metrics: throughput and power.

5.2 LDBC Graphalytics

LDBC Graphalytics (31) is a benchmark for graph analysis platforms. Graphalytics contains six deterministic algorithms: Breadth-first search (BFS), PageRank (PR), Weakly connected components (WCC), Community detection using label propagation (CDLP), local clustering coefficient (LCC), and single-source shortest path (SSSP). It uses the LDBC

SNB Datagen and Graph500 data generator as input data. While Graphalytics also uses the SNB Datagen, it uses scale factors with different clustering coefficients, e.g., SF100 with clustering coefficients 0.05 and 0.15. The difference between Graphalytics and SNB is that SNB mainly targets DBMSs capable of running graph workloads. In contrast, Graphalytics targets platforms running analysis algorithms that may not be easily modeled as database queries, for example extracting global metrics from the graph. Another difference is that SNB-I provides the driver to the user and the user only needs to implement a client using an interface.

5.3 LSQB: Large-Scale Subgraph Query Benchmark

LSQB (38), Large-scale Subgraph Query Benchmark, focuses on DBMS join performance by using subgraph matching, equivalent to multi-way joins between vertices and edges tables on ID attributes. The benchmark uses the LDBC SNB data generator as the data set. Its workload consists of 9 read-heavy global queries and lacks update operations. Six queries match basic graph patterns (equivalent to joins on edge tables) and three look at complex graph patterns (necessitating anti-join and outer join operations) (3). The key differences between LSQB and LDBC SNB-I are that SNB-I uses seeded queries (e.g., starting a query on a person node specified by a query parameter), includes update operations and covers path queries in its workload.

5.4 LinkBench

LinkBench (5) is a synthetic benchmark created by Facebook based on the production database traces of their systems. The benchmark aims to simulate real-world database workloads for social applications. The queries used are point reads by primary key, create, delete and update operations, selection of ranges by ID, type, and timestamp, and aggregation queries like counting the number of friends. In addition, LinkBench only defines one type of node, simplifying the benchmark and creating edges during bulk load, not considering underlying correlations usually found in social networks since the queries do not directly assess performance with such structures.

5.5 GDB-test

Lissandri et al. (35) introduce a microbenchmark targeting graph databases. It is derived from the LDBC SNB Interactive workload by decomposing the complex read queries and

look at the basic operators. The operators are evaluated using 35 queries, in the categories load, create, read, update, delete and traversals. The data set that is used in the benchmark is, besides the LDBC SNB, real data from different domains, e.g., Yeast, a protein interaction network. The evaluation metric of the benchmark is the disk space used, the data loading and query execution times. A limitation of this microbenchmark is the small scale of the graphs.

5.6 TPC (Transaction Processing Performance Council)

5.6.1 TPC-C

TPC-C is an OLTP benchmark to compare database platforms running medium complexity transaction processing workload (64), having five concurrent transaction types: selects, updates, inserts, deletes, joins and non-unique selects. It simulates the activity of a wholesale supplier, handling new orders, payments, order status, delivery and stock levels. (64). A feature of TPC-C compared to previous TPC benchmarks is that it contains skew in the access patterns for certain relations in the data. In addition, the benchmark workload can be partitioned across multiple nodes executing the benchmark (34).

5.6.2 TPC-H

TPC-H (66) is a decision support benchmark consisting of ad-hoc queries and concurrent updates. The ad-hoc querying workload simulates users sending individual queries that are not known to the database in advance, such that the database administrator cannot optimize the database systems. The benchmark has 22 read-only queries and two refresh functions, inserting and removing rows from tables (45).

5.6.3 TPC-DS

TPC-DS (65) is a decision support benchmark that targets OLAP DBMSs. In contrast to TPC-H, TPC-DS includes 99 queries covering the whole simulation data set, by having ad-hoc, reporting, data mining and iterative OLAP queries (46). It also addresses deficiencies found in TPC-H. The synthetically generated data used in TPC-H scaled linearly leading to unrealistic scenarios where the data is unskewed, imposing little challenge on statistic collection and optimal plan generation (39). Furthermore, the third normal form schema does not sufficiently stress the differences in indexing techniques and query optimizers (39),

which is changed with TPC-DS using a snowflake schema. Lastly, the refresh operations were not testing the capabilities of a DBMS under realistic data maintenance operations.

5.7 YCSB

The Yahoo Cloud Serving Benchmark (YCSB) (14) targets OLTP DBMSs in the cloud, providing 5 workloads with different percentages of operations, ranging from 50% reads and 50% updates for the update heavy workload to 95–100% read operations. It primarily targets serving In addition, a workload including mostly scan operations is included. YCSB comes with a Java client that generates the data and executes the workload. The client can also be extended to support other DBMSs and workloads.

Part II

Distributed Driver Design

6

Tools for Distributed Benchmarking

To design a distributed driver, we look for tools and frameworks to support and simplify the implementation of the driver. Specifically, we investigate tools that enable the deployment of a distributed driver that is relatively easy to set up as well as maintainable: changing part of the driver should not lead to a significant effort. Additionally, we select tools and libraries that enable simplifying the implementation of the driver clients to reduce complexity. As a starting point for the selection, we use the Cloud Native Compute Foundation to investigate existing frameworks and tools.

6.1 Cloud Native Compute Foundation

The Cloud Native Compute Foundation (CNCF) (12), part of the Linux Foundation, is an organization that encourages the adoption of cloud native technologies by providing resources to projects to support adoption as well as a technology *landscape*[1], providing an overview of available technologies. Examples of areas where projects are supported are containerization, programmable infrastructure, CI/CD, storage and logging. CNCF defines cloud native technologies to enable loosely coupled systems that are resilient, manageable and observable.[2]

6.2 Distributed Frameworks

Several frameworks are available for distributed applications, such as Kubernetes and Akka.io.

[1] `https://landscape.cncf.io`
[2] `https://github.com/cncf/toc/blob/main/DEFINITION.md`

6.2.1 Kubernetes

Originally developed at Google under the name Borg (68), Kubernetes is an open-source system for automating deployment, scaling, and managing containerized applications. It runs on most platforms and has multiple distributions which target small local instances (minikube, k3s, Docker Kubernetes, kubeedge). This enables Kubernetes to run from lightweight edge/Internet-of-Things hardware to large-scale hosted solutions on major public clouds like Azure AKS (Azure Kubernetes Service), AWS EKS (Elastic Kubernetes Service), and Google GKE (Google Kubernetes Engine).

Kubernetes schedules the containerized application onto a node or cluster depending on the resource requirements. The smallest schedulable unit in Kubernetes is called a Pod: a group of one or more containers that share storage and network resources and share the same specification for how to run the containers. This enables, for example, the deployment of a containerized application together with a log collector for that application. The containers in a Pod are always co-located on the same node and co-scheduled, analogous to applications that would have run on the same (virtual) machine. An example of this is shown in Figure 6.1.

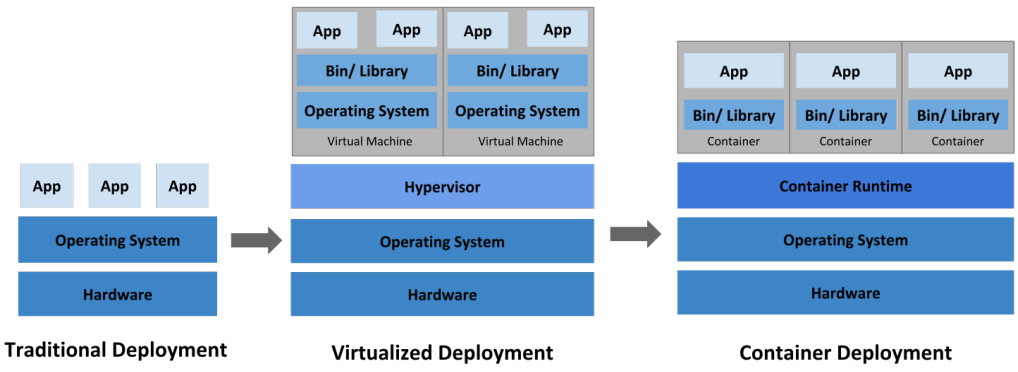

Figure 6.1: Mapping from virtual machine with applications to Kubernetes. Image from (6)

A Pod can be scheduled in multiple ways:

- **Deployments:** Describes the desired state of Pods with for example ReplicaSets, a new state for the Pods or scaling up a deployment.
- **StatefulSets:** Like a deployment, but maintains a sticky identity for each of the Pods: the Pods are created using the same specification, but are not interchangeable.

- **Replicaset:** A replicaset maintains a stable set of replica Pods at any time to guarantee the availability of a specified number of identical Pods.
- **Daemonsets:** ensures that all nodes of the Kubernetes cluster run a copy of a Pod. This is useful for example with Pods handling logging, monitoring or cluster storage.
- **Jobs:** creates one or more Pods and will continue to retry until successful termination. When a failure occurs, it will retry.

6.2.2 Akka.io

Akka.io[1] is a distributed framework using the actor model. The actor model has *actors*, which can send messages to other actors, create new actors and change the local state. This model aims to handle two challenges in concurrent programs: delegate tasks to other threads with encapsulation without blocking and handling of service faults. To pass tasks to other threads, actors send messages asynchronously to each other, delegating work to each other. A message does not have a return value: when a task needs to return a value, it is done through a message to avoid the original sender from blocking until the return value. Akka.io provides a set of libraries to use this programming model, as well as libraries to enable communication with a variety of protocols such as HTTP, gRPC and streaming data.

6.2.3 Message Passing Interface (MPI)

MPI (20) is a message passing specification used in parallel computing. It provides routines to enable communication between threads on the same machine or threads on multiple different machines. MPI defines data types to support data to be sent in heterogeneous environments. In addition, it provides the specification of functions to enable point-to-point communication between processes (send/receive) as well as collective communication functions, such as broadcast and scatter to send the data to multiple threads and gather to receive the data or reduce functions to aggregate results. MPI is commonly used in High Performance Computing (HPC) environments. Since MPI is a specification, multiple implementations exist, such as OpenMPI[2] and Intel MPI[3].

[1] `https://akka.io`
[2] `https://www.open-mpi.org`
[3] `https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html`

7

Blueprint for a Distributed Benchmark Framework

The distributed driver allows horizontally scaling the driver and target distributed database systems for benchmarking. However, there are challenges in creating a distributed driver. First, the workload needs to be distributed *evenly* across the clients, such that the work a client receives has roughly the same runtime as other clients. We elaborate on the distribution of the workload in Section 7.1. Second, the clients need to communicate the completion time to the other clients. This is required since the interactive driver uses dependency tracking for the update streams and complex queries. The design of the distributed driver, including communication and setup, is discussed in Section 7.2.

7.1 Distributing the Benchmark Workload

The complex read queries are initialized during the startup of the driver and scheduled during the benchmark, with the start time of the first query depending on the earliest scheduled update operation. The complex operations are created in an infinite incrementing loop in the single node driver (see Section 2.6.5.2), with the scheduled time depending on the frequency of a query. While the update streams can be partitioned using round-robin (the same strategy used in the Hadoop Datagen) to distribute scheduled operations evenly across clients, the complex read streams are challenging to partition since the (expected)[1] runtimes per query can differ a lot, leading to unbalanced runtimes for clients in

[1]The expected runtimes refers to the frequency assigned to a complex read query: if the frequency number is lower, that query is scheduled more often to have an approximately equal amount of total runtime for queries with higher frequency numbers.

a distributed setting. To partition the complex reads, we will discuss four strategies.

1. Serialize the entire workload, then partition We first create the workload stream together with the update streams, and afterward we partition the workload over the number of clients. Each client then gets an individual stream to execute until exhausted. An advantage of this approach is that the logic in the client is simple: it only needs to execute each operation from the stream and communicate the dependency time every T_{safe} time. Additionally, this partitions the substitution parameters preventing a query from being executed by all the clients with the same parameters. A disadvantage is the distribution of queries over the clients: this approach assumes the runtimes are partitioned equally while that may not be the case since a partition can be assigned multiple short-running queries. To check how the queries would be distributed over multiple clients, we try the round-robin approach and frequency-based balancing. With frequency-based balancing, we partition the queries using their frequencies as a weight, since the frequency is tied to the average runtime of a query[1]. Each client then receives a complex read stream with approximately the same average runtime.

2. Operation skipping based on node ID This approach, each node is assigned an ID n based on the total number of nodes N. Then, each node will create the workload stream and execute the $n, n+N, \ldots n+iN$ operation in the stream. This way the workload does not have to be generated and serialized before execution. In principle, the execution is the same as serializing the workload and partitioning it afterward using a round-robin approach. However, this approach does not require sending a serialized workload, making distribution easier at the expense of a client with more complex logic to generate the complex reads.

3. Recalculate frequencies and assign different start times The frequencies are used to determine the scheduling distance between executed query operations per type. When the frequencies are not changed in a distributed setup and each client N will execute the complex reads as scheduled with a single node, each query type will be executed N times more (the number of clients). This will result in the benchmark being unbalanced between complex, short and update operations. If the frequencies are multiplied by N, the scheduled time between operations of the same query time increases. However, this still does not prevent having all the clients execute the same query at once. After scheduling

[1]The frequencies are set so each query is equally important in the benchmark (see Section 2.6.5.2)

the complex streams, the client skips n amount of complex streams and shifts the start time to the first time in the complex stream, which will ensure that each client starts with a different query and the start times in the stream are shifted so each client will execute a different complex query.

4. Central operation queue with operation batches Using this approach lets a coordinator create the workload and create batches with complex read queries, which can then be fetched by a client when requesting new operations. Each operation batch can then be balanced by the coordinator with approximately the same amount of complex reads with the same frequencies. Each batch contains multiple complex reads and a client can request a new one before it runs out of operations. This way, the client does not have to wait for new operations, creating a potential bottleneck. A disadvantage of this approach is that a coordinator needs to be created: the batches need to be large enough to hide potential latency between the coordinator and client. This also requires an additional channel of communication to get the batches, next to the communication of the dependency time. Because of the additional complexity and the communication overhead, this approach will not be taken into consideration.

7.2 System Design

To create the distributed driver, we use the Kubernetes framework (see Section 6.2). We choose this framework above others since this is framework can be run on most public cloud providers as well as on-premise setups. Additionally, it allows us to separate required services to simplify the implementation and maintenance.

In Section 2.6.5, the components of the driver are shown. To design the distributed driver, we identify the following required components:

- **Client:** The client should be able to execute queries to the SUT and communicate their completion time with the other clients. The client has two components: the first component is the driver that executes the query and logs the result. The second component is specific to the SUT and contains the required logic to create a connection with the SUT, the query templates and individual query handlers that are responsible for converting fields, e.g., datetimes, to the format used by the SUT.

- **Communication service:** The clients should be able to communicate their latest completion time with each other.
- **Coordinator:** The clients need to be deployed by a coordinator that keeps the state of the clients: (*startup*, *running*, *failed*, *completed*) and assigns each client an update stream and the number of operations to execute.
- **Logging & Observability:** The clients produce logs of the executed queries, their throughput and the current state. The logs are then collected and should be observable by the user. This gives insight into the state during the benchmark as well as access to logs for evaluation/debugging.

The system design is based on the proposed design in (29), where a benchmark environment is mapped to Kubernetes. However, in our design, the responsibility of deploying the SUT is left to the user as the SUT can run in a container, but can also be deployed outside the cluster. Furthermore, the distributed interactive benchmark driver requires multiple clients to communicate with each other. In the following subsections, we will dive into the details of each component. The components are chosen from the CNCF (12) Landscape[1], providing an overview of available tools.

7.2.1 Logging & Observability

During benchmark execution, the state of the benchmark as well as the results must be communicated to the user. This is done by monitoring each client's logs and storing them in a database where the results can be queried by a dashboard. To collect the logs, Fluentd[2] is used. Fluentd is an open-source data collector, part of the CNCF member projects. It collects the logs as a stream, then writes the data to Prometheus[3], a monitoring system and time series database. This allows for collecting the logs in a central place and querying them. The logs can then be queried using the Grafana[4] dashboard, which allows for querying and visualizations of the data. The logging workflow is shown in Figure 7.1.

7.2.2 Communication

In between the clients, their local Completion time needs to be communicated to the other clients for dependency tracking. The clients, therefore, communicate with a global com-

[1] `https://landscape.cncf.io`
[2] `https://www.fluentd.org`
[3] `https://prometheus.io`
[4] `https://grafana.com`

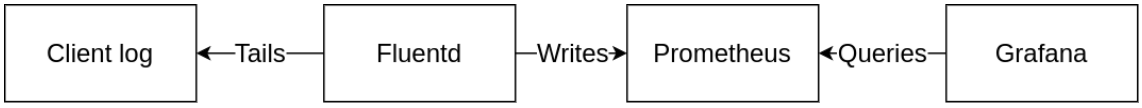

Figure 7.1: Workflow of the logging for the distributed driver

pletion time service, which is a separate Pod in the Kubernetes cluster. The service keeps track of all the local completion times and clients can request the global completion time. The communication is done through a *service mesh*[1]. A service mesh is a way to manage the service-to-service communication, in this case, the communication of all the clients with the completion time service. Istio[2] is a service mesh providing this communication service. It runs alongside each Pod, also known as *sidecars*. The communication logic is therefore separated from the client, simplifying the implementation of the client. Using the service mesh can also help in tracing requests and debugging the communication between the clients. If more fine-grained tracing is required, services like Jaeger[3] can be used to observe networking communication.

Another consideration is the time between the scheduled update operation and their dependent date and the relation with the latency of the driver clients with the completion time service. The ΔT between scheduled and dependent operations have a minimum of 10,000 milliseconds, assuming $TCR = 1.0$, not taking the execution time of the update into account. When a lower value for the TCR is used, the ΔT will be smaller. Therefore, for communication, the latency should be lower than $TCR \times 10000 + T_{max}$, where T_{max} is the execution time of a query specific to the system.

7.2.3 Deployment

To deploy the benchmark on a Kubernetes cluster in a reproducible way, a Kubernetes Operator is created. An Operator allows to define Custom Resource Definitions regarding the driver components and to deploy them using the Kubernetes API. The Operator contains the definitions required for starting the environment with the specified number of clients and their properties, which can be configured by the user in a YAML file. After deployment, the operator assigns each client an update stream partition and starts the benchmark once all clients are up and running. This requires the update streams to be accessible for

[1] `https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh`
[2] `https://istio.io`
[3] `https://www.jaegertracing.io`

the clients and already partitioned. The Operator contains logic to observe the state of the clients. Each client communicates its state whether it is initializing, started, stopped or failed. The distributed driver architecture for Interactive v2.0 is given in Figure 7.2.

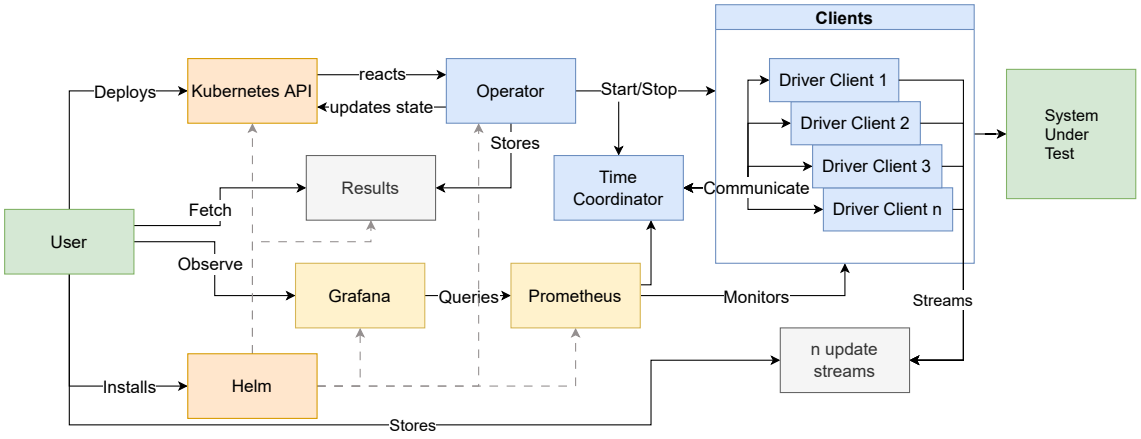

Figure 7.2: Distributed driver architecture. *Orange* shows the elements that are present in the Kubernetes cluster, *Yellow* are the external components that need configuration, *Blue* are the components specifically for the distributed driver.

8

Evaluation of Partitioning Strategies

In this section we evaluate partitioning strategies discussed in Section 7.1. Three workload partitioning strategies are evaluated by simulating the complex read scheduler: round-robin distribution, frequency-based distribution and frequency recalculation using SF10. As approximation for the relative runtimes per partition, we use the mean runtimes from the results of Neo4j in Section 4.2.2. While the runtimes per database system is different and not known prior to benchmarking a system, it serves as a practical example how the partitioning of the workload affects client total runtimes.

8.1 Complex Read Partitioning

The scheduling of complex reads is explained in Section 2.6.5. To simulate the scheduler from the interactive driver, we use Algorithm 2. This produces a stream of complex queries with their scheduled time with the following assumptions:

- The time compression ratio of the scheduled queries equals $TCR = 1.0$
- The amount of scheduled operations is 10,000
- The query frequencies and interleaves used are from the SF10 workload

In Figure 8.1 we show the relative runtime per thread for three different partitioning strategies: round-robin distribution of queries, frequency-based partitioning and frequencies based on the number of clients. With round-robin and frequency-based partitioning, we observe that when the number of clients increases, the relative difference in runtimes per client increases: with 32 clients, the client with the lowest runtime has 2% of the total runtime and the client with the highest runtime approximately 4%, double the runtime

compared to the lowest. When running the benchmark, this means that the client with the lowest runtime is idle half of the benchmark duration. When the frequencies are recalculated based on the number of threads, we observe a more even distribution in runtimes compared to the two previous strategies. The difference between the smallest and highest runtime is 0.27% when using 32 clients compared to 2.09% and 2.06% for round-robin and frequency-based partitioning respectively. Table 8.1 shows the amount of scheduled queries per partitioning strategy per type when scheduling 10,000 operations and selecting the first hour of the scheduled queries. round-robin and frequency-based partitioning have the same amount of scheduled operations per query type. This is expected since the operation stream is created before partitioning. When the frequencies are recalculated, the number of scheduled queries is lower for queries with low frequencies, e.g., Query 8 and Query 11. Reason for this difference is the additional time between queries because of the increased frequencies. However, this difference can be overcome by adjusting the frequencies for these queries.

Algorithm 2 Workload distribution: Complex read scheduler

Input: G, T_{start}, $T_{interleave}$, $N_{operations}$, f_{query}
Output: Scheduled operations $S_{operations}$ of length $N_{operations}$
  Initialize operation stream $S_{operations}$
  Initialize list with first scheduled time per query $L_{T_{scheduled}}$ using T_{start}
  for N in $N_{operations}$ **do**
    Get earliest $T_{scheduled}$ query from L
    Add query to operation stream
    Increase $T_{scheduled}$ for query with $T_{interleave} \times f_{query}$
  end for

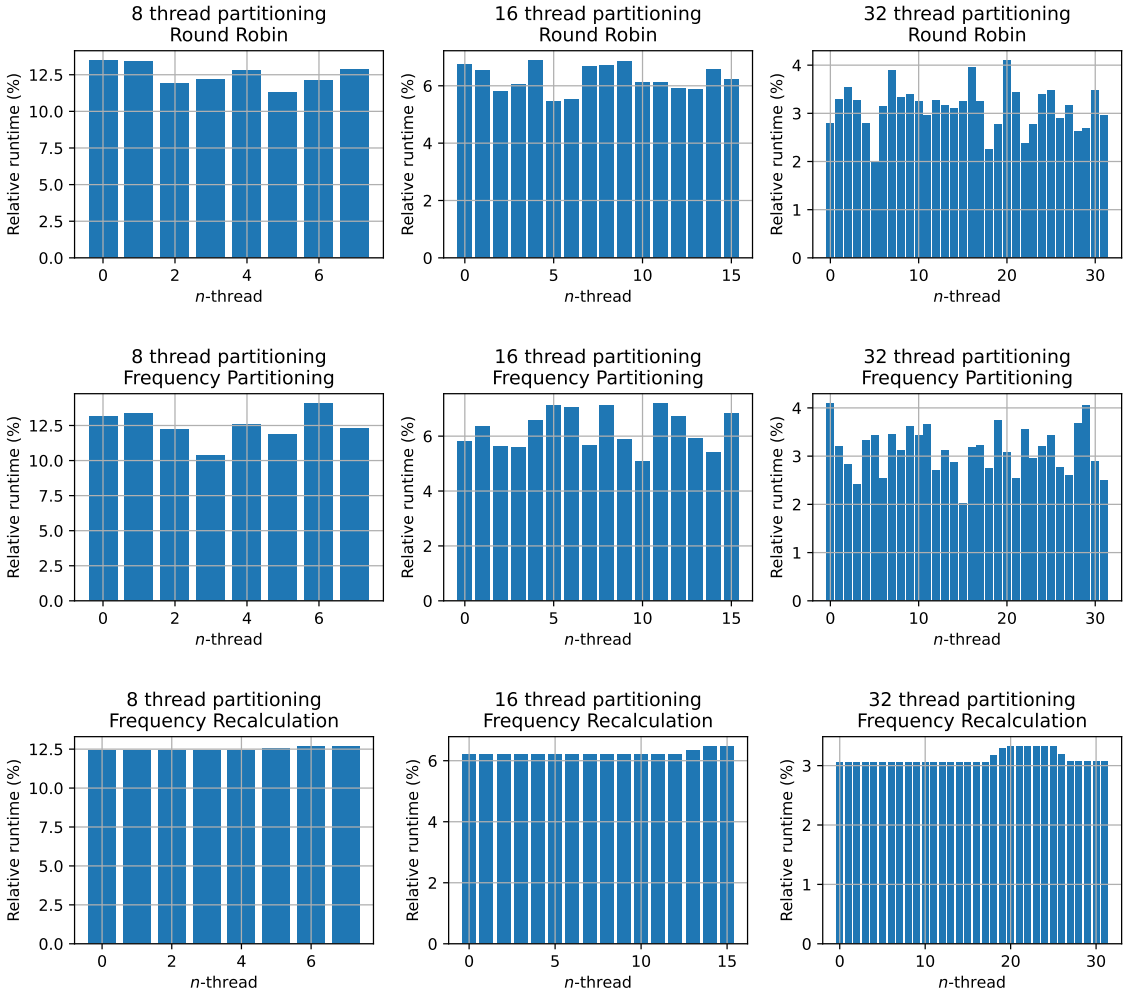

Figure 8.1: Workload distribution: relative runtimes per thread for three different partitioning strategies.

Query	Round-robin	Frequency-based	Frequency Recalculation
LdbcQuery1	854	854	849
LdbcQuery2	600	600	599
LdbcQuery3a	120	120	120
LdbcQuery3b	120	120	120
LdbcQuery4	617	617	614
LdbcQuery5	336	336	336
LdbcQuery6	94	94	88
LdbcQuery7	411	411	408
LdbcQuery8	1,481	1,481	1,474
LdbcQuery9	77	77	72
LdbcQuery10	634	634	630
LdbcQuery11	1,169	1,169	1,166
LdbcQuery12	505	505	504
LdbcQuery13a	584	584	584
LdbcQuery13b	584	584	584
LdbcQuery14a	284	284	280
LdbcQuery14b	284	284	280

Table 8.1: Workload distribution: scheduled counts per query for each workload partitioning strategy by scheduling 10000 queries and selecting the first hour of scheduled queries.

9

Related Work on Distributed Benchmark Frameworks

This section provides information about current work on distributed benchmark frameworks.

9.1 Orchestrating DBMS Benchmarking in the Cloud with Kubernetes

Erdelt (17) provides a way to organize DBMS benchmarking in the cloud using a Kubernetes cluster. Their motivation is to create a benchmark workflow that can be executed in parallel to speed up the benchmarking process for different scales and multiple hardware configurations when using a heterogeneous cluster.

They identified the following components typically found in DBMS benchmarks and mapped those components to Kubernetes objects. To begin, the Loader (bulk) loads the initial data stored in a persistent volume on the cluster into a Master, the DBMS under test, and Workers, optional parts of the DBMS if it supports distributed setup. This Loader is an external shell script that only runs during initialization and therefore does not run on the cluster. The Maintainer, which executes update queries to update the contents of the database, is a separate thread outside the cluster, similar to the Loader. When executing the benchmark, the orchestrator deploys multiple Pods for the query executors in a queue with different configurations per Pod. While there is a benchmark configuration in the queue, the benchmark will execute until termination. The benchmark results are collected by a metrics collector, running as a Pod on the cluster, and stored in persistent storage

where a dashboard can access the data.

While their work describes a mapping between benchmark and Kubernetes components, several things differ in their approach. First, they designed a benchmark workflow on Kubernetes to parallelize the entire workflow, not the number of clients. As a result, the benchmark driver (Query Executor) can use multiple threads but uses one container; therefore, the driver is not distributed. Second, the results exclude the use of the Maintainer, which is responsible for updating the data in the DBMS. In the design, the Maintainer was not part of the driver but a separate script triggered outside the cluster.

9.2 Reproducible Benchmarking of Cloud-Native Applications with the Kubernetes Operator Pattern

Henning et al. (29) describes a way to implement a benchmark capable of benchmarking cloud-native (12) applications. Cloud-native applications are typically several separate processes running in isolated containers, scheduled on different computer nodes, requiring application-level configuration, network setup, and integration of different storage systems, making benchmarking such applications complex. They provide a role and data model for describing benchmarks and their execution to simplify defining, distributing, and executing benchmarks. They identified two roles. First is the Benchmark Designer's role, who knows the SUTs and how to interpret result metrics, creating a benchmarking tool or artifact. The Benchmarker role executes the benchmark and intends to compare and rank different SUTs.

They propose to use the Kubernetes Operator pattern to implement a benchmark, where combining the knowledge of operating Kubernetes with domain knowledge is possible. The operational knowledge can be implemented using Kubernetes Custom Resource Definitions (CRDs). Two CRDs are defined: one for describing the benchmark configuration, the SUT benchmarking method, which is stateless since they can execute arbitrarily often, and one for execution, which has a state like pending, running, finished or failed. Metrics are collected using Prometheus, a monitoring service, which a dashboard like Grafana can access. This approach separates the setup of the benchmark and the execution and bundles in one file the experimental setup, manageable with Kubernetes.

Their work targets applications in general and not specifically DBMSs, and the paper only describes a model with the required components. The author gives an implementation for benchmarking the scalability of distributed stream processing engines. However, they combined the definition of the SUT and benchmark tool into the CRDs description, which requires changes to make a benchmark tool using this approach generally applicable to other SUTs.

9.3 Is It Safe To Dockerize My Benchmark?

Grambow et al. (25) found that dockerizing[1] a DBMS benchmark has a measurable and non-constant influence on the results. The experiments ran 30 runs, using the YCSB workload, with four different setups in AWS (using m3.medium and m3.large machines): no dockerization, only dockerized benchmarking client, dockerized SUT, and full dockerization. They found that, when comparing the dockerized results with the non-dockerized results, the average latency ranged from -1% 7% when only the client is dockerized, -3% 7% when only the SUT is dockerized, and -2% 12% when everything was dockerized. In addition, they noted that the differences and variance in performance by Docker are likely the effects of performance variances in the underlying virtual machine. Finally, they conclude that the actual numbers are too unreliable to measure system performance. However, in Iosup et al. (32), cloud services have variating performance based on yearly and daily time patterns, influencing the virtual machine's performance. Since the experiments with the Docker containers are executed on a virtual machine in the cloud susceptible to these performance variabilities, the benchmark results with the dockerized results are not conclusive, making it difficult to interpret the results. Therefore, the effect of dockerizing the SUT should be evaluated on a bare-metal machine.

9.4 DIAMetrics

DIAMetrics (15) is a benchmark framework developed at Google. Being developed to test multiple internal query engines (F1, Procella, Dremel), the framework provides a way to test these with different workloads to serve different use cases, either synthetic benchmarks like TPC-H or production-like data that is anonymized using the framework. In addition, it can extract workloads from query engine logs for custom benchmark generation. However,

[1] dockerization is a container based on a Docker image, using the docker runtime

the framework can only parallelize the workload runner by sending the same workload of all targeted systems for execution in parallel, making this different from the driver for LDBC SNB-I, where the driver must execute the queries in parallel while tracking the dependencies.

9.5 PEEL

PEEL is an open-source framework to define, execute, analyze and share experiments (9). It can automatically orchestrate experiments and handles the systems setup, configuration, deployment, tear-down and cleanup, and the collection of the logs. In addition, it allows for hardware-independent specification. It currently supports JVM-based systems. The authors tested PEEL with a supervised machine-learning workload on Apache Spark and Apache Flink. The main goal is to automate all the intermediate steps a user needs to do when benchmarking these systems:

- setting up the distributed file system (HDFS),
- ingesting and transforming the data set used for the benchmark,
- submitting the benchmark workload as a job, metrics collection, and tear-down of the SUT, together with cleanup of the file system.

10

Future Work

The work in this thesis project touched upon several parts of the LDBC Social Network Benchmark suite to include deletes, improve scalability, and create a reference distributed benchmark driver. In the following, we propose several points for future work.

Integrating update stream creation into the Spark Datagen The current update stream output of the Spark Datagen (Datagen version `v0.5.1`) is not suitable for the Interactive workload since it does not contain the dependency time required in the driver to prevent insertion/deletion of entities that have dependencies. We circumvent this deficiency with an external Python/SQL script, selecting the data from the raw temporal graph and selecting the dependency time. However, Datagen should be altered to include this property in the update streams.

Investigate events in Spark data set The Spark data set shows a skew of events towards the end of the simulation timeframe with almost no noticeable spiking events during the simulation. To remove the skew towards the end of the simulation window and to introduce spiking events, the Spark Datagen's configuration and method of generating temporal attributes should be investigated.

Temporal factor tables The current factor tables lack temporal information, leading to parameter curation with inaccurate information. While not all queries are affected by this problem, queries that need to traverse the friendship network or the number of friends' messages can benefit from this information, improving parameter curation. In addition, the effects of implicit deletes need to be taken into account as well: once a person is

deleted from the network, their comments, posts, and forums are deleted as well, affecting the respective factor tables.

Implementing distributed driver This thesis only provides a design for a distributed driver, investigating the tools and frameworks to use as well as a suitable partitioning strategy for complex read queries. Therefore, an implementation of the distributed driver is left out of scope of this thesis.

Integration of the SNB BI workload into the driver The refactoring of the driver enables more straightforward implementation of other workloads, such as the SNB Business Intelligence workload, which uses the same data set. By implementing this, users can execute the BI benchmark with concurrent updates, which is currently unavailable in the Python driver of the BI workload, despite being allowed by the benchmark specification (1, Section 7.5). Extending the driver with this workload makes it more versatile and solves the complex endeavor of creating a separate BI driver that handles concurrent update streams with dependencies.

11

Conclusion

In this thesis, we modernized the LDBC SNB Interactive Workload by improving its scalability, feature coverage, and usability. The LDBC SNB Interactive Driver code has undergone a major refactor to improve performance and usability. We added support for the LDBC SNB Spark Datagen to the driver, enabling the use of larger scale factors and update streams with delete operations. This thesis created a temporal parameter generation and added support for temporal substitution parameters in the driver, allowing to query freshly inserted entities during the benchmark. Additionally, we provided a distributed driver design. Lastly, we created a reference implementation for Microsoft SQL Server. The following provides answers to the research questions of this thesis.

How can deletions be integrated into the Social Network Benchmark Interactive? To integrate deletions in the SNB Interactive driver, we changed the data set from the LDBC SNB Hadoop Datagen to the LDBC SNB Spark Datagen, producing a temporal graph with delete operations. This change required a separate, batched update stream converter since the update streams from the Spark Datagen do not disclose dependency time information. In addition, the update stream formats changed, requiring changes in the driver to support loading Parquet files. Using a batched loader, the driver can support the loading of update streams for larger scale factors and handles the parallelization of the update streams internally. The new loader removes the need for users to partition the update streams first and control parallelization with the amount of update stream files. The delete queries are integrated into the driver and the reference implementations. Our analysis of the characteristics of the data sets produced by the Spark Datagen (Section 4.2) revealed several differences regarding the number of events and missing flashmob events in

the Spark data set. The changed characteristics require additional investigation to ensure that the benchmark sufficiently covers the challenges of bursting temporal events.

How to generate parameters with similar behavior to the query template with the inclusion of inserted and deleted nodes and edges? Including deletes in the Interactive benchmark affects the substitution parameters used in the query templates, which can result in unpredictable query runtimes. We solve this problem by introducing temporal parameter curation, which allows generating parameters in daily batches with approximately the same runtime behavior. However, the factor table statistics are affected by the dynamic nature of the temporal graph, giving inaccurate information. To mitigate this problem for the path queries, we created a path curation that provides valid 4-hop paths between two persons. Experiments show that the temporal parameter curation gives a smaller standard deviation in runtimes compared to the v1.0 parameter generation for only 6 of the 14 queries, which is related by the inaccuracy of the used factor tables. The new parameter generator version demonstrates better scalability than the parameter generation in v1.0, achieving up to 100× better runtimes for SF1,000. We show that the new parameter generation can generate parameters for SF3,000 and SF10,000. The temporal parameters required changes in the query execution to make the driver aware of when to execute a parameter by using a start and expiry date for each parameter given by the temporal parameter generation.

What effect does the inclusion of deletion operations have on the performance of the systems under test? Delete operations in the Interactive benchmark can have performance implications for the SUT. Experiments using Neo4j and Umbra show that deletions for scale factors 10, 30, 100, and 300 do not show significant differences in runtimes when removing an edge. However, when a person is removed, the cascading effect of the deletion increases when using larger scale factors. In addition, experiments done on DBMS X demonstrate that cascading deletes after the deletion of a node can have significant performance implications on certain systems.

How can the Interactive benchmark driver be made distributed? We provide a distributed driver design based on Kubernetes, allowing deployment in different environments. Experiments show that partitioning the complex read operations such that runtimes per client are best done by recalculating the query frequencies. This allows for balanced total runtimes per client. However, Round-robin and frequency-based partitioning showed

that some clients are idle half the time; therefore, they are unsuitable as a partitioning strategy. Further research is required to determine an optimal partitioning strategy for the update streams.

References

[1] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba Pey, Norbert Martínez, József Martaon, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasić, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. **The LDBC Social Network Benchmark**, 2021. vii, 1, 9, 14, 18, 21, 22, 24, 37, 38, 39, 40, 55, 65, 90, 109, 110, 111, 112, 113, 114

[2] Renzo Angles, Marcelo Arenas, Pablo Barcelo, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, Oskar van Rest, and Hannes Voigt. **G-CORE: A Core for Future Graph Query Languages**. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1421–1432, New York, NY, USA, 2018. Association for Computing Machinery. Available from: `https://doi.org/10.1145/3183713.3190654`. 1

[3] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. **Foundations of Modern Query Languages for Graph Databases**. *ACM Comput. Surv.*, **50**(5):68:1–68:40, 2017. Available from: `https://doi.org/10.1145/3104031`. 66

[4] Renzo Angles, Peter Boncz, Josep Larriba-Pey, Irini Fundulaki, Thomas Neumann, Orri Erling, Peter Neubauer, Norbert Martinez-Bazan, Venelin Kotsev, and Ioan Toma. **The Linked Data Benchmark Council: A graph and RDF industry benchmarking effort**. *ACM SIGMOD Record*, **43**(1):27–31, 2014. 1

[5] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. **Linkbench: a database benchmark based on the face-**

book social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1185–1196, 2013. 2, 66

[6] The Kubernetes Authors. **Overview of Kubernetes**, 2023. Available from: https://kubernetes.io/docs/concepts/overview/. ix, 72

[7] Bradley R Bebee, Daniel Choi, Ankit Gupta, Andi Gutmans, Ankesh Khandelwal, Yigit Kiran, Sainath Mallidi, Bruce McGaughy, Mike Personick, Karthik Rajan, et al. **Amazon Neptune: Graph Data Management in the Cloud.** In *International Semantic Web Conference (P&D/Industry/BlueSky)*, 2018. 1, 10, 11

[8] Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michal Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. **Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries**. *CoRR*, **abs/1910.09017**, 2019. Available from: http://arxiv.org/abs/1910.09017. 61

[9] Christoph Boden, Alexander Alexandrov, Andreas Kunft, Tilmann Rabl, and Volker Markl. **PEEL: A framework for benchmarking distributed systems and algorithms**. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 9–24. Springer, 2017. 88

[10] Peter Boncz, Thomas Neumann, and Orri Erling. **TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark**. In Raghunath Nambiar and Meikel Poess, editors, *Performance Characterization and Benchmarking*, pages 61–76, Cham, 2014. Springer International Publishing. 2, 8

[11] Angela Bonifati, Irena Holubová, Arnau Prat-Pérez, and Sherif Sakr. **Graph Generators: State of the Art and Open Challenges**. *ACM Comput. Surv.*, **53**(2):36:1–36:30, 2021. Available from: https://doi.org/10.1145/3379445. 14

[12] CNCF. **CNCF Cloud Native Definition v1.0**, Jun 2018. Available from: https://raw.githubusercontent.com/cncf/toc/main/DEFINITION.md. 71, 78, 86

[13] World Wide Web Consortium et al. **RDF 1.1 concepts and abstract syntax**. 2014. 8, 9

[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. **Benchmarking Cloud Serving Systems with YCSB**. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery. Available from: https://doi.org/10.1145/1807128.1807152. 68

[15] Shaleen Deep, Anja Gruenheid, Kruthi Nagaraj, Hiro Naito, Jeff Naughton, and Stratis Viglas. **Diametrics: benchmarking query engines at scale**. *ACM SIGMOD Record*, **50**(1):24–31, 2021. 87

[16] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoč, Mingxi Wu, and Fred Zemke. **Graph Pattern Matching in GQL and SQL/PGQ**. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 2246–2258, New York, NY, USA, 2022. Association for Computing Machinery. Available from: https://doi.org/10.1145/3514221.3526057. 1, 2, 10

[17] Patrick K. Erdelt. **Orchestrating DBMS Benchmarking in the Cloud with Kubernetes**. In Raghunath Nambiar and Meikel Poess, editors, *Performance Evaluation and Benchmarking*, pages 81–97, Cham, 2022. Springer International Publishing. 85

[18] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. **The LDBC social network benchmark: Interactive workload**. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 619–630, 2015. 1, 2, 17, 21, 24

[19] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. **KÙZU Graph Database Management System**. In *CIDR*, 2023. Available from: https://www.cidrdb.org/cidr2023/papers/p48-jin.pdf. 1

[20] Message P Forum. **MPI: A Message-Passing Interface Standard**. Technical report, USA, 1994. 73

[21] Apache Foundation. **Apache Parquet**, 2022. Available from: https://parquet.apache.org/. 19

[22] Apache Software Foundation. **Apache TinkerPop**, 2022. Available from: https://tinkerpop.apache.org. 11

[23] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. **Cypher: An evolving query language for property graphs**. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1433–1445, 2018. 1

[24] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. **Cypher: An evolving query language for property graphs**. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1433–1445, 2018. 9

[25] Martin Grambow, Jonathan Hasenburg, Tobias Pfandzelter, and David Bermbach. **Is It Safe to Dockerize My Database Benchmark?** In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, pages 341–344, New York, NY, USA, 2019. Association for Computing Machinery. Available from: https://doi.org/10.1145/3297280.3297545. 87

[26] Jim Gray. **Database and Transaction Processing Performance Handbook**, 1993. 7, 14

[27] The PostgreSQL Global Development Group. **About Postgres**, 2022. Available from: https://www.postgresql.org/about/. 13

[28] Andrey Gubichev and Peter Boncz. **Parameter Curation for Benchmark Queries**. In Raghunath Nambiar and Meikel Poess, editors, *Performance Characterization and Benchmarking. Traditional to Big Data*, pages 113–129, Cham, 2015. Springer International Publishing. 2, 21, 23

[29] Sören Henning, Benedikt Wetzel, and Wilhelm Hasselbring. **Reproducible Benchmarking of Cloud-Native Applications with the Kubernetes Operator Pattern**. 2021. 78, 86

[30] Torsten Hoefler and Roberto Belli. **Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results**. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–12, 2015. 7

[31] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, et al. **LDBC Graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms**. *Proceedings of the VLDB Endowment*, **9**(13):1317–1328, 2016. 65

[32] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. **On the performance variability of production cloud services**. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 104–113. IEEE, 2011. 87

[33] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. **Query optimization through the looking glass, and what we found running the join order benchmark**. *The VLDB Journal*, **27**(5):643–668, 2018. 8

[34] Scott T. Leutenegger and Daniel Dias. **A Modeling Study of the TPC-C Benchmark**. *SIGMOD Rec.*, **22**(2):22–31, jun 1993. Available from: `https://doi.org/10.1145/170036.170042`. 67

[35] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. **Beyond macrobenchmarks: microbenchmark-based graph database evaluation**. *Proceedings of the VLDB Endowment*, **12**(4):390–403, 2018. 2, 66

[36] Frank McSherry, Michael Isard, and Derek G Murray. **Scalability! But at what COST**. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015. 14

[37] Memgraph. **Open Source Graph Database**, 2022. Available from: `https://memgraph.com/`. 10

[38] Amine Mhedhbi, Matteo Lissandrini, Laurens Kuiper, Jack Waudby, and Gábor Szárnyas. **LSQB: A Large-Scale Subgraph Query Benchmark**. In *Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, GRADES-NDA '21, New York, NY, USA, 2021. Association for Computing Machinery. Available from: `https://doi.org/10.1145/3461837.3464516`. 66

[39] Raghunath Othayoth Nambiar and Meikel Poess. **The Making of TPC-DS.** In *VLDB*, **6**, pages 1049–1058, 2006. 67

[40] Neo4j. **Neo4j**, Oct 2021. Available from: https://neo4j.com/. 1, 8

[41] Thomas Neumann and Michael J. Freitag. **Umbra: A Disk-Based System with In-Memory Performance**. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. Available from: http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf. 13

[42] Ontotext. **About GraphDB**, 2022. Available from: https://graphdb.ontotext.com/documentation/10.0/about-graphdb.html. 11

[43] David A. Patterson. **For better or worse, benchmarks shape a field: Technical perspective**. *Commun. ACM*, **55**(7):104, 2012. Available from: http://doi.acm.org/10.1145/2209249.2209271. 8

[44] Minh-Duc Pham, Peter Boncz, and Orri Erling. **S3g2: A scalable structure-correlated social graph generator**. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 156–172. Springer, 2012. 14

[45] Meikel Poess and Chris Floyd. **New TPC Benchmarks for Decision Support and Web Commerce**. *SIGMOD Rec.*, **29**(4):64–71, dec 2000. Available from: https://doi.org/10.1145/369275.369291. 67

[46] Meikel Poess, Bryan Smith, Lubor Kollar, and Paul Larson. **TPC-DS, Taking Decision Support Benchmarking to the next Level**. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, page 582–587, New York, NY, USA, 2002. Association for Computing Machinery. Available from: https://doi.org/10.1145/564691.564759. 67

[47] PostGIS. **About PostGIS**, 2022. Available from: https://postgis.net. 13

[48] Mark Raasveldt, Pedro Holanda, Tim Gubner, and Hannes Mühleisen. **Fair benchmarking considered difficult: Common pitfalls in database performance testing**. In *Proceedings of the Workshop on Testing Database Systems*, pages 1–6, 2018. 7

[49] Mark Raasveldt and Hannes Mühleisen. **DuckDB: An Embeddable Analytical Database**. In *SIGMOD*, pages 1981–1984. ACM, 2019. Available from: https://doi.org/10.1145/3299869.3320212. 35

[50] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases: new opportunities for connected data*. O'Reilly Media, Inc., 2015. 8

[51] Marko A. Rodriguez. **The Gremlin Graph Traversal Machine and Language (Invited Talk)**. In *Proceedings of the 15th Symposium on Database Programming Languages*, DBPL 2015, page 1–10, New York, NY, USA, 2015. Association for Computing Machinery. Available from: https://doi.org/10.1145/2815072.2815073. 11

[52] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. **The graph story of the SAP HANA database**. In Volker Markl, Gunter Saake, Kai-Uwe Sattler, Gregor Hackenbroich, Bernhard Mitschang, Theo Härder, and Veit Köppen, editors, *Datenbanksysteme für Business, Technologie und Web (BTW) 2037*, pages 403–420, Bonn, 2013. Gesellschaft für Informatik e.V. 2, 10

[53] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. **The ubiquity of large graphs and surprising challenges of graph processing**. *Proceedings of the VLDB Endowment*, **11**(4):420–431, 2017. 1, 2

[54] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A Boncz, et al. **The future is big graphs: a community view on graph processing systems**. *Communications of the ACM*, **64**(9):62–71, 2021. 1

[55] Margo I. Seltzer, David Krinsky, Keith A. Smith, and Xiaolan Zhang. **The Case for Application-Specific Benchmarking**. In Peter Druschel, editor, *Proceedings of The Seventh Workshop on Hot Topics in Operating Systems, HotOS-VII, Rio Rico, Arizona, USA, March 28-30, 1999*, pages 102–109. IEEE Computer Society, 1999. Available from: https://doi.org/10.1109/HOTOS.1999.798385. 7

[56] Supreeth Shastri, Vinay Banakar, Melissa Wasserman, Arun Kumar, and Vijay Chidambaram. **Understanding and Benchmarking the Impact of GDPR on Database Systems**. *Proc. VLDB Endow.*, **13**(7):1064–1077, 2020. Available from: http://www.vldb.org/pvldb/vol13/p1064-shastri.pdf. 3

[57] Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. **NetworKit: A tool suite for large-scale complex network analysis**. *Netw. Sci.*, **4**(4):508–530, 2016. Available from: https://doi.org/10.1017/nws.2016.20. 45, 56

[58] Michael Stonebraker and Lawrence A Rowe. **The design of Postgres**. *ACM Sigmod Record*, **15**(2):340–355, 1986. 13

[59] Dávid Szakállas. **Speeding up LDBC SNB Datagen**, Jun 2020. Available from: https://ldbcouncil.org/post/speeding-up-ldbc-snb-datagen/. 19

[60] Gábor Szárnyas, Arnau Prat-Pérez, Alex Averbuch, József Marton, Marcus Paradies, Moritz Kaufmann, Orri Erling, Peter Boncz, Vlad Haprian, and János Benjamin Antal. **An early look at the LDBC social network benchmark's business intelligence workload**. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, pages 1–11, 2018. 65

[61] Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter A. Boncz. **The LDBC Social Network Benchmark: Business Intelligence Workload**. *Proc. VLDB Endow.*, **16**(4):877–890, 2022. Available from: https://www.vldb.org/pvldb/vol16/p877-szarnyas.pdf. 1, 65

[62] Daniel ten Wolde, Tavneet Singh, Gábor Szárnyas, and Peter Boncz. **DuckPGQ: Efficient property graph queries in an analytical RDBMS**. In *CIDR*, 2023. Available from: https://www.cidrdb.org/cidr2023/papers/p66-wolde.pdf. 2

[63] Yuanyuan Tian, En Liang Xu, Wei Zhao, Mir Hamid Pirahesh, Sui Jun Tong, Wen Sun, Thomas Kolanko, Md. Shahidul Haque Apu, and Huijuan Peng. **IBM Db2 Graph: Supporting Synergistic and Retrofittable Graph Queries Inside IBM Db2**. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 345–359, New York, NY, USA, 2020. Association for Computing Machinery. Available from: https://doi.org/10.1145/3318464.3386138. 2

[64] TPC. **TPC-C Homepage**. Available from: https://www.tpc.org/tpcc/default5.asp. 67

[65] TPC. **TPC-DS Homepage**. Available from: `https://www.tpc.org/tpcds/default5.asp`. 67

[66] TPC. **TPC-H Homepage**. Available from: `https://www.tpc.org/tpch/default5.asp`. 67

[67] TuGraph. **TuGraph**, 2022. Available from: `https://www.tugraph.org`. 8, 10, 12

[68] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. **Large-scale cluster management at Google with Borg**. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015. 72

[69] World Wide Web Consortium W3C. **SPARQL 1.1 Overview**, 2013. Available from: `https://www.w3.org/TR/2013/REC-sparql11-query-20130321/`. 11

[70] Jack Waudby, Benjamin A. Steer, Karim Karimov, József Marton, Peter A. Boncz, and Gábor Szárnyas. **Towards Testing ACID Compliance in the LDBC Social Network Benchmark**. In Raghunath Nambiar and Meikel Poess, editors, *Performance Evaluation and Benchmarking - 12th TPC Technology Conference, TPCTC 2020, Tokyo, Japan, August 31, 2020, Revised Selected Papers*, **12752** of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2020. Available from: `https://doi.org/10.1007/978-3-030-84924-5_1`. 28

[71] Jack Waudby, Benjamin A Steer, Arnau Prat-Pérez, and Gábor Szárnyas. **Supporting Dynamic Graphs and Temporal Entity Deletions in the LDBC Social Network Benchmark's Data Generator**. In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, pages 1–8, 2020. 3, 33

[72] Jim Webber. **A Programmatic Introduction to Neo4j**. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, page 217–218, New York, NY, USA, 2012. Association for Computing Machinery. Available from: `https://doi.org/10.1145/2384716.2384777`. 9, 12

[73] Min Wu, Xinglu Yi, Hui Yu, Yu Liu, and Yujue Wang. **Nebula Graph: An open source distributed graph database**. *CoRR*, **abs/2206.07278**, 2022. Available from: `https://doi.org/10.48550/arXiv.2206.07278`. 1

Appendix

Appendix A

Short Read Generation

Read query	S1	S2	S3	S4	S5	S6	S7
Q1	⊗	⊗	⊗				
Q2	⊗	⊗	⊗	⊗	⊗	⊗	⊗
Q3	⊗	⊗	⊗				
Q7	⊗	⊗	⊗	⊗	⊗	⊗	⊗
Q8	⊗	⊗	⊗	⊗	⊗	⊗	⊗
Q9	⊗	⊗	⊗	⊗	⊗	⊗	⊗
Q10	⊗	⊗	⊗				
Q11	⊗	⊗	⊗				
Q12	⊗	⊗	⊗				
Q14	⊗	⊗	⊗				
IS2	⊗	⊗	⊗	⊗	⊗	⊗	⊗
IS3	⊗	⊗	⊗				
IS5	⊗	⊗	⊗				
IS6	⊗	⊗	⊗				
IS7	⊗	⊗	⊗	⊗	⊗	⊗	⊗

Table A.1: Short read queries executed after read query

Appendix B

Parameter Curation Factor Table Selection

Query 6

Given a start Person with ID `$personId` *and a Tag with name* `$tagName`*, 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 (1)*

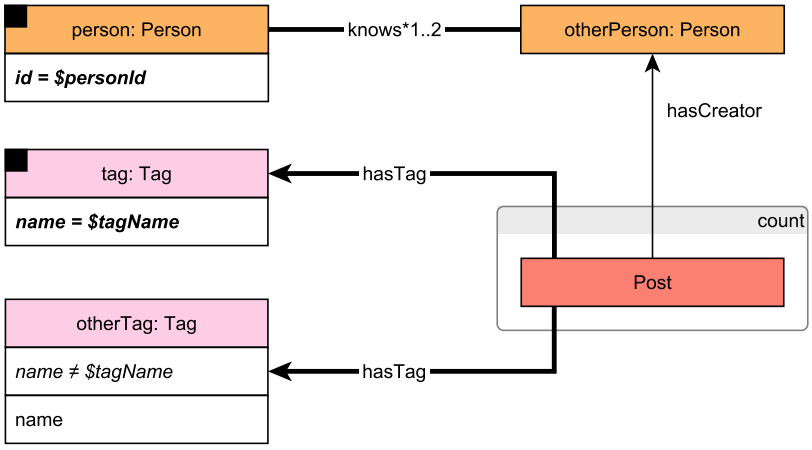

Figure B.1: Pattern of LDBC SNB Interactive Complex Read 6

For Query 6 (Figure B.1), each `$personId`, we want a similar amount of friends of friends and `$tagNames`. We therefore select from the `personNumFriendsOfFriendsOfFriends` factor table person IDs with similar number of friends of friends. For the tag names, we

use the `tagNumPersons` factor table, which contains for each tag their frequency in the graph. We select tags with similar frequencies.

Query 7

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

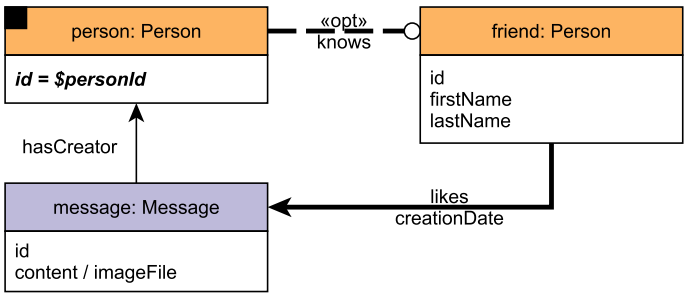

Figure B.2: Pattern of LDBC SNB Interactive Complex Read 7

For Query 7 (Figure B.2) we need to select persons with similar amount of direct friendships from the `personNumFriends` table. While there is a factor table with statistics how many likes a person has received, we opted not to use this table for two reasons: 1) the frequency only gives us information during the whole simulation window, 2) giving likes to a post or comment in the benchmark is one of the most frequent update operations, therefore frequency given in the factor table can be too inaccurate to use.

Query 8

Given a start Person with ID `$personId`*, find the most recent Comments that are replies to Messages of the start Person. Only consider direct (single-hop) replies, not the transitive (multi-hop) ones. Return the reply Comments, and the Person that created each reply Comment.* (1)

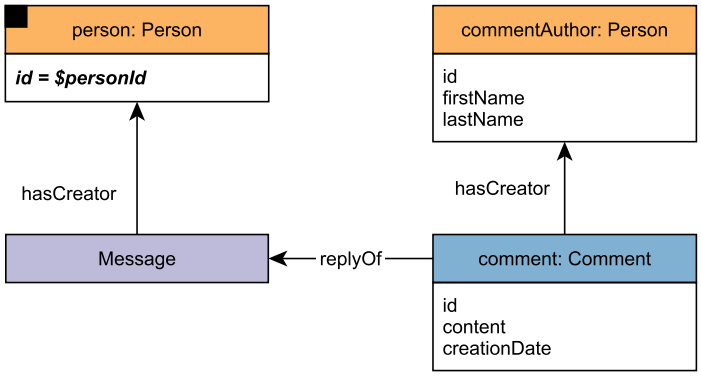

Figure B.3: Pattern of LDBC SNB Interactive Complex Read 8

For Query 8 (Figure B.3), we select the number of direct comments a person has using the `personNumFriendComments` factor table.

Query 9

Given a start Person with ID `$personId`*, find the most recent Messages created by that Person's friends or friends of friends (excluding the start Person). Only consider Messages created before the given* `$maxDate` *(excluding that day). (1)*

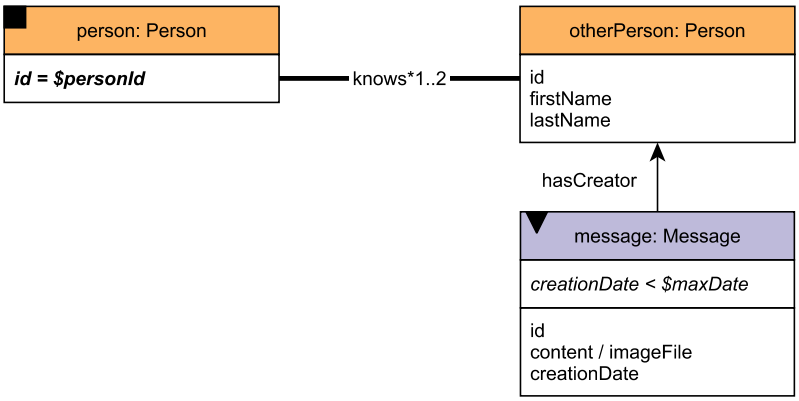

Figure B.4: Pattern of LDBC SNB Interactive Complex Read 9

For Query 9 (Figure B.4), we select the person Ids using the `numFriendsOfFriends`. To select a suitable `$maxDate`, we use the information from the `creationDayNumMessages` that gives us a date with the number of messages available in the network.

Query 10

> *Given a start Person with ID* ***`$personId`****, find that Person's friends of friends (foaf) - excluding the start Person and his/her immediate friends -, who were born on or after the 21st of a given* ***`$month`*** *(in any year) and before the 22nd of the following month. Calculate the similarity between each friend and the start person.* (1)

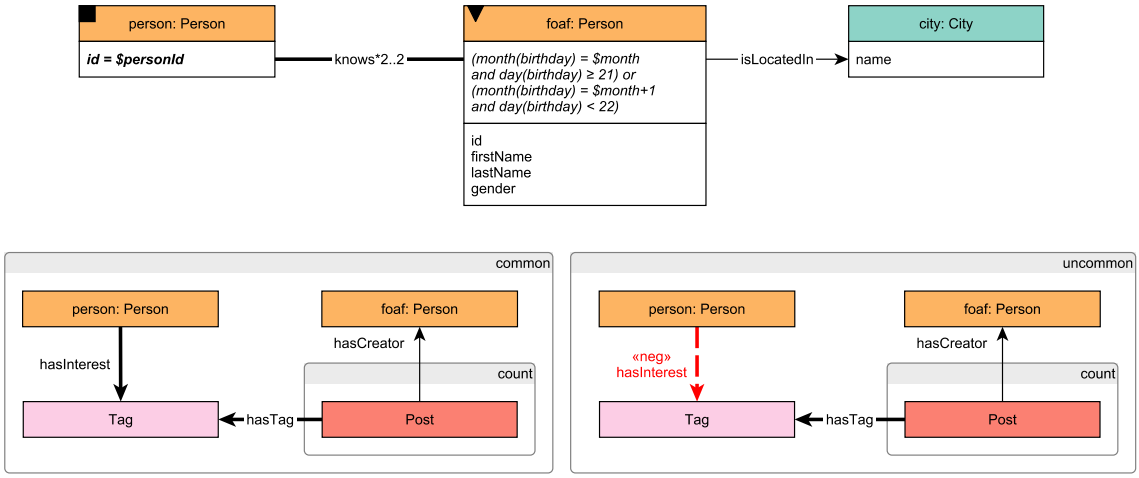

Figure B.5: Pattern of LDBC SNB Interactive Complex Read 10

For Query 10 (Figure B.5), we select the person IDs from the `personNumFriendOfFriends` factor table to select similar friend of friends frequencies. To select a month, we generate a series of numbers in the range 1 to 12.

Query 11

Given a start Person with ID `$personId`*, find that Person's friends and friends of friends (excluding start Person) who started working in some Company in a given Country with name* `$countryName`*, before a given date (*`$workFromYear`*).* (1)

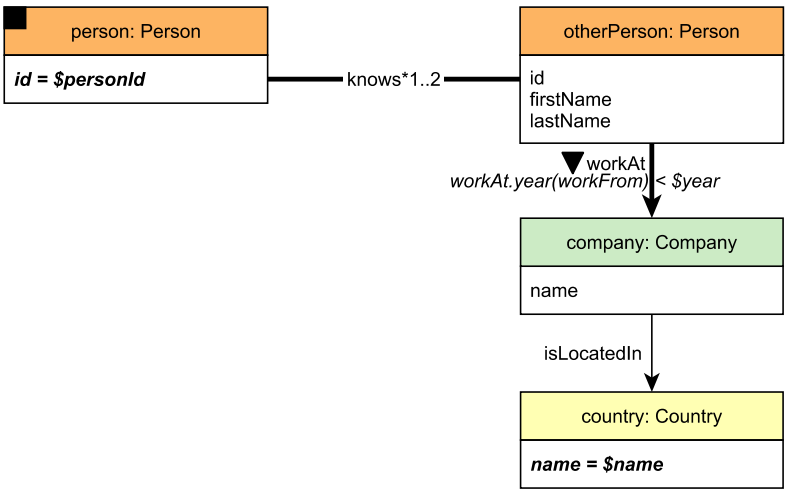

Figure B.6: Pattern of LDBC SNB Interactive Complex Read 11

For Query 11 (Figure B.6), we need the number of friends and friend of friends from the `numFriendsOfFriends` factor table, together with selecting a country name from the `countryNumPersons`. The year is generated, giving a range from 1998 until 2013 (end of the benchmark simulation window).

Query 12

Given a start Person with ID `$personId`*, find the Comments that this Person's friends made in reply to Posts, considering only those Comments that are direct (single-hop) replies to Posts, not the transitive (multi-hop) ones. Only consider Posts with a Tag in a given TagClass with name* `$tagClassName` *or in a descendent of that TagClass. Count the number of these reply Comments, and collect the Tags that were attached to the Posts they replied to, but only collect Tags with the given TagClass or with a descendant of that TagClass. Return Persons with at least one reply, the reply count, and the collection of Tags.* (1)

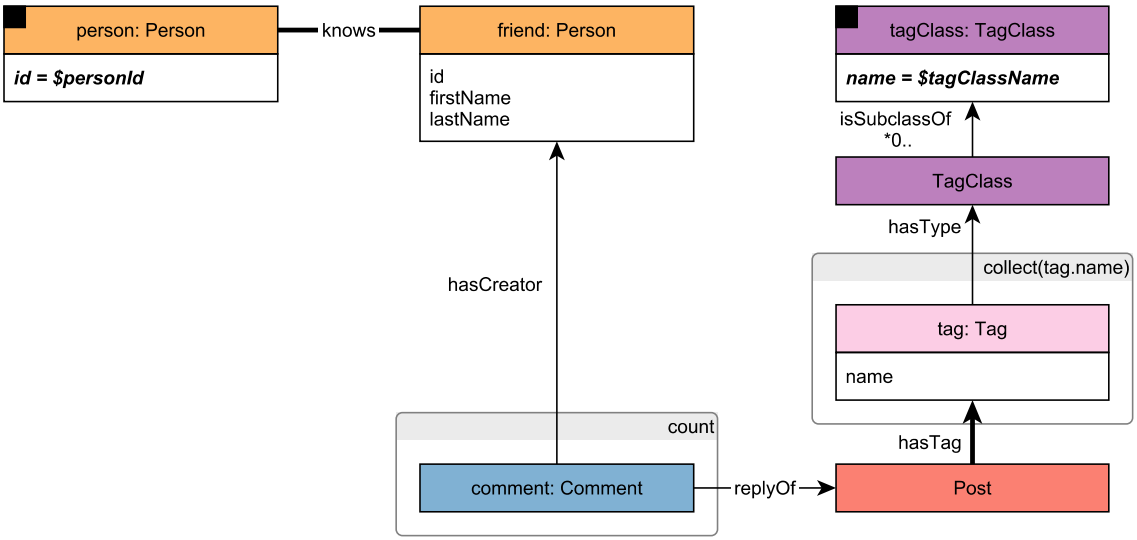

Figure B.7: Pattern of LDBC SNB Interactive Complex Read 12

For Query 12 (Figure B.7) we select the number of direct friends from the `personNumFriends` table. The tagClassName is selected from the `tagClassNumTags` table, which gives the frequency per tagClassName.

Appendix C

Parameter Selection

Selecting Windows

In this example, we find a window of personIds with a number of friends of friends close to each other in cardinality. We use the `personNumFriendsOfFriendsOfFriends` factor table from SF10. The figures are limited in the x and y axis for simplicity. First, we sort the factor table on the number of friends of friends to see the distribution. The result is shown in Figure C.1. Second, of the sorted column we take the difference between the neighbors and group the person IDs with a difference below a specified threshold. The resulting groups are shown in Figure C.2. Lastly, we select the window with a minimum number of person IDs, in this example 25, and then select the window with the smallest standard deviation. The result is shown in Figure C.3.

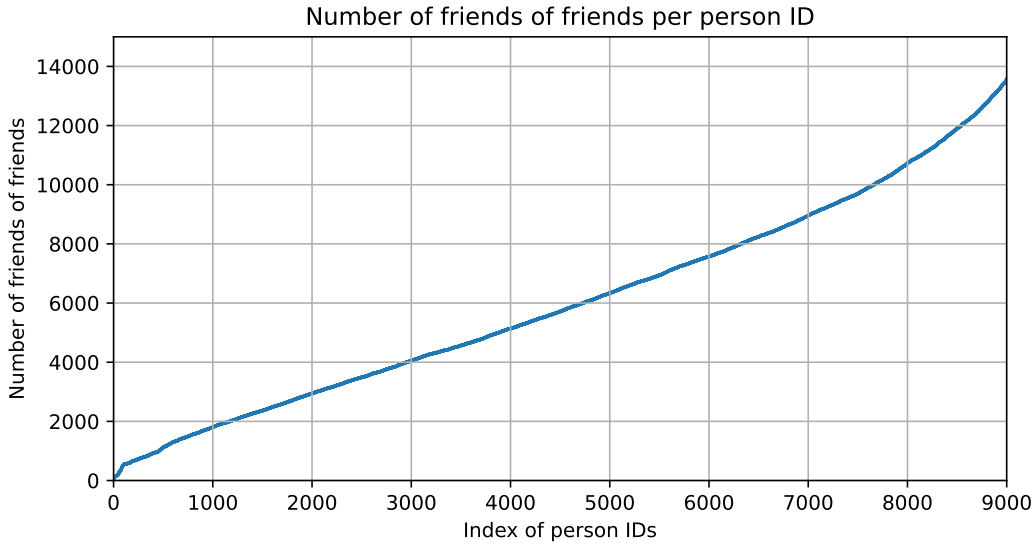

Figure C.1: Distribution of the number of friends of friends per person ID.

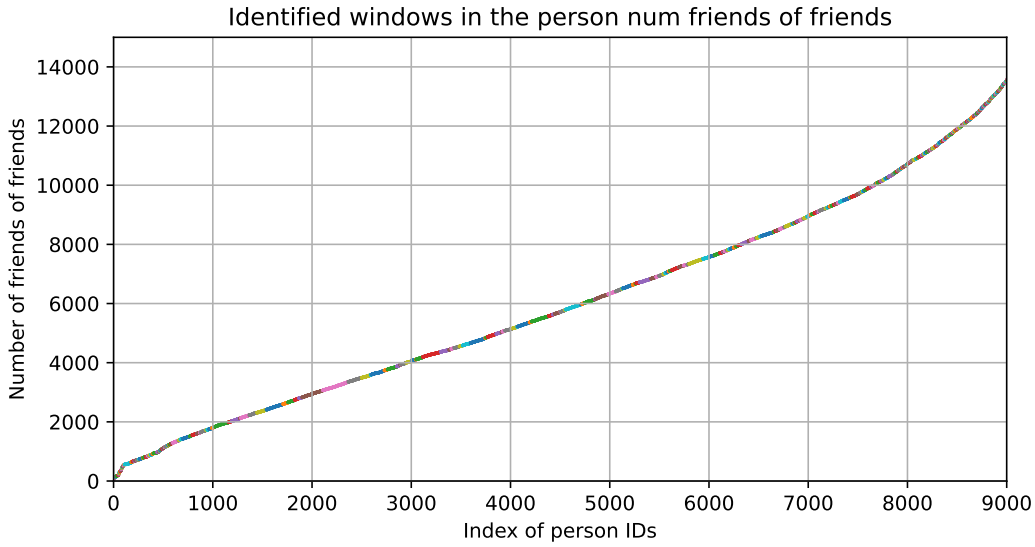

Figure C.2: Grouped person IDs in the distribution of number of friends of friends

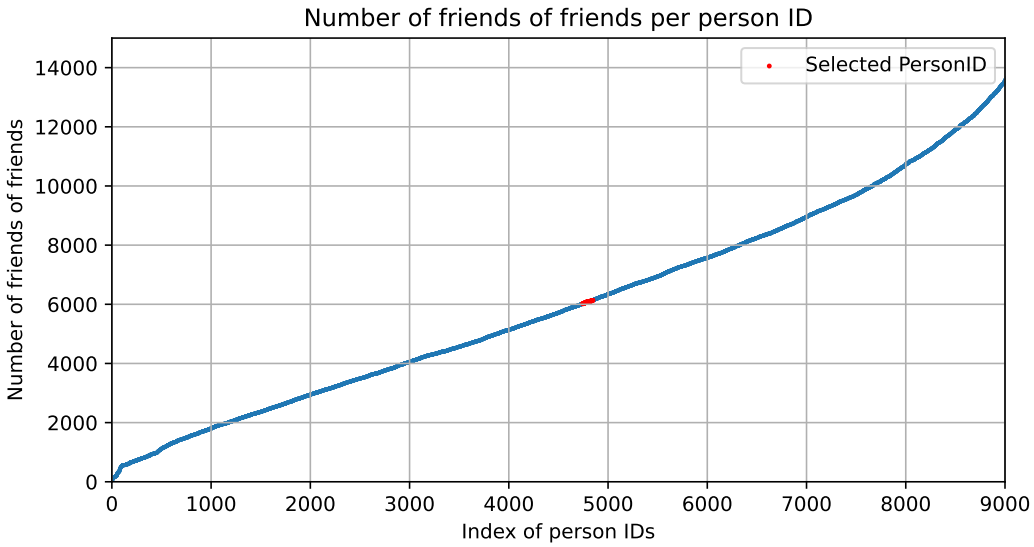

Figure C.3: Selected window of person IDs in the distribution of number of friends of friends

Appendix D

Parameter Curation Query 1 Example

This is a query that uses the parameters that are preselected using window functions and selects the ones that are valid for a given day. We then take the cartesian product between the personIDs and firstNames.

```
SELECT personId  AS 'personId',
       firstName AS 'firstName',
       useFrom   AS 'useFrom',
       useUntil  AS 'useUntil'
FROM
    (
        SELECT Person1Id AS personId,
               creationDate AS useFrom,
               deletionDate AS useUntil
          FROM personNumFriendsOfFriendsOfFriendsSelected
         WHERE deletionDate - INTERVAL 1 DAY > :date_limit_filter
           AND creationDate + INTERVAL 1 DAY < :date_limit_filter
         ORDER BY md5(Person1Id)
         LIMIT 50
    ),
    (
        SELECT firstName,
          FROM personFirstNamesSelected
         ORDER BY md5(Person1Id)
         LIMIT 20
    )
ORDER BY md5(concat(personId, firstName, useUntil, useFrom))
LIMIT 500
```

Listing D.1: Example of daily parameter selection for Query 1.

Appendix E

Scaling the Data sets to SF30,000

To scale up the driver, the Spark Datagen should generate larger scale factors whose sizes are determined by the following definition: the scale factors denote the size of the generated CSV files in GiB, e.g., the SF10 datase consists of approximately 10 GiB of CSV files. At the beginning of this thesis work, the Spark Datagen was already capable of producing large graphs with billions of edges but its scale factors were not yet precisely calibrated to ensure that the size of the generated datases conform to this rule. Datagen takes the number of persons in a graph as input parameter for each scale factor. To approximate the amount of persons in the graph for scale factors 3,000 and up, we used polynomial interpolation. As input parameters, the known number of persons per scale factor (up to SF300) is given, shown in Table E.1.

SF	1	3	10	30	100	300	1,000	3,000	10,000	30,000
numPerson	10,620	25,870	70,800	175,950	487,700	1,230,500	3,505,000	9,232,000	27,200,000	77,000,000

Table E.1: Number of persons in the graph for a given scale factor.

$$N_{persons} = 7.522 \cdot 10^{-13}x - 3.363 \cdot 10^{-8}x^2 + 3.729 \cdot 10^{-4} - 1.289x^3 + 4.425 \cdot 10^{3}x^4 + 2.638 \cdot 10^{4}x^5 \tag{E.1}$$

The prediction, based on the known values up to SF300, is done using a 5^{th}-degree polynomial. The result is a polynomial approximating the number of persons in the graph, shown in Equation E.1. This formula was used to approximate the number of persons for scale factors 1,000 to 30,000.

E. SCALING THE DATA SETS TO SF30,000

Changelog

Driver

- Added scale factor frequencies to the driver, users can now select the scale factor within the client properties file instead of providing the query frequencies and update interleaves
- Refactored validation parameter serialization to include query name
- Moved from CSV reader to Parquet based loader using DuckDB
- Refactored substitution parameter and update streams readers to use a single class
- Integrated support for temporal substitution parameters in operation handlers
- Merged read and write threads to use a threadpool instead of separate threads for writes
- Integrated batched update stream reader with batch size parameter to minimize memory usage when using larger scale factors
- Changed scheduled validation rule from a maximum of 10 late operations per query to 5% per query
- Upgraded libraries and driver from Java 8 to Java 11
- Added variants for queries 3, 13, and 14
- Refactored test classes to split long running tests into smaller ones to enable more granular testing
- Upgraded JUnit 4 to JUnit (Jupiter) 5
- Parallelized testing in CI/CD pipeline and in JUnit

- Created Python script that uses DuckDB and SQL to generate update streams with dependency time
- Added temporal parameter curation script and SQL parameter generation query templates
- Added temporal path curation script using networkit to replace people4Hops factor table

Implementations

- Added reference implementation for SQL Server
- Add delete query skeletons in the Java client code to all reference implementations
- Fixed connection pooling for Postgres, Umbra and SQL Server by using HikariCP
- Moved QueryStore class to common project to generalize query building for implementations
- Added docker container loaders for Postgres, Umbra and SQL Server together with docker-compose SUT setup
- Update Postgres loader from psycopg2 to ppsycopg3