Master Thesis

Temporal Graph Data Management for in-memory Database Systems

Author(s):
Rohr, Philipp

Publication Date:
2015

Permanent Link:
https://doi.org/10.3929/ethz-a-010412113

Rights / License:
In Copyright - Non-Commercial Use Permitted
Master’s Thesis Nr. 128

Systems Group, Department of Computer Science, ETH Zurich

Temporal Graph Data Management for in-memory Database Systems

by

Philipp Rohr

Supervised by

Dr. Martin Kaufmann
Prof. Dr. Donald Kossmann

Abstract

Today’s database landscape consists of the traditional relational database systems which in the last years have started to support temporal aspects. These database systems allow for keeping versions of tuples and making queries for data at a certain Point in Time or from a Time Range. The database landscape also contains NoSQL databases like document stores or graph databases. These graph databases do not yet support any kind of temporal aspects like the ones mentioned before. This thesis provides a basis for graph database developers to improve their products in temporal aspects.

To be able to build, compare and evaluate temporal graph databases, general use cases must be available. In this thesis we identify three dimensions of use cases for temporal graph databases, namely Queries Over Attributes, Graph Related Queries / Graph metrics and Time Related Queries. In these dimensions, we identify several classes of query types which can be combined, and which result in concrete queries on temporal graph databases. Having these use cases, we build a scalable data generator that generates data settled in a social network example. The frameworks we use for implementing the benchmark allow us to run the queries and measure execution times on different temporal database systems.

We measured the query performance of three commercial products: Neo4j, which is a graph database with no explicit temporal support, and two relational databases, a disk-based row store with native support for Point in Time and Time Range queries, and an in-memory column store with native support for Point in Time queries. The experiments show that the databases with native support for temporal queries outperform Neo4j, for which the temporal support had to be simulated using its on-board tools. Neo4j scores when it comes to graph traversal queries, like Shortest Paths between nodes, which cannot be implemented with traditional relational databases. But these graph traversal queries on Neo4j are limited, because they do not support additional constraints, which would be necessary to support Point in Time and Time Range queries. We simulate these query types with detours, which result in poor execution times.

Conclusively we can say that today’s graph databases are not yet at the point to support temporal queries in a productive environment, and optimizations in this direction are necessary in order to achieve better performance.
Acknowledgements

I thank Prof. Dr. Donald Kossmann and Dr. Martin Kaufmann who gave me the opportunity to write this thesis in ETH’s Systems Group. It was very interesting to dive into Martin’s field of research, of which I had not heard much up to the start of the thesis. Martin was always there when I had questions about any details of temporal databases and was a great sparring partner in discussions about engineering and implementation. Thanks also go to Prof. Dr. Peter Fischer and Io Taxidou from the Albert-Ludwigs-Universität Freiburg who invited us for a short trip to Freiburg to discuss their view of the topic. A huge thank you goes to Christine Zeller who invested hours in correcting my English. Without her language skills and patience, this thesis would be littered with passive sentences and a lot less majestic plural clauses. Last but not least I hug my wife Sandra who always supported me and encouraged me to finish my master’s studies.
Contents

1 Introduction ................................................. 5
  1.1 Motivation ............................................. 5
  1.2 Problem Statement ..................................... 5
  1.3 Contribution .......................................... 5
  1.4 Outline ................................................. 6

2 Related Work ................................................ 7
  2.1 Temporal Relational Databases ......................... 7
  2.2 Temporal Graphs ....................................... 8
  2.3 Graph Databases ....................................... 8
  2.4 Benchmarks ............................................. 9

3 Background .................................................. 10
  3.1 Examples Setting ....................................... 10

4 Temporal Graphs ............................................. 12
  4.1 Graphs .................................................. 12
  4.2 Temporal Graph ......................................... 12
    4.2.1 Temporal Subgraph ................................. 13
    4.2.2 Fixed Temporal Graph .............................. 14
    4.2.3 Static Graph ...................................... 15
  4.3 Storing a Temporal Graph ............................... 15
    4.3.1 Temporal Relational Database ...................... 15
    4.3.2 Graph Databases .................................. 17

5 Use Cases .................................................... 21
  5.1 Query Types ............................................. 21
    5.1.1 Queries Over Attributes ............................ 21
    5.1.2 Graph Related Queries / Graph Metrics ............ 21
    5.1.3 Time Related Queries ............................... 22
  5.2 Examples Setting ........................................ 22
  5.3 Non Temporal Queries - Queries on a Static Graph .... 22
  5.4 Point in Time Queries - Queries on a Fixed Temporal Graph 25
  5.5 Time Range Queries - Queries on a Temporal Subgraph..... 25
    5.5.1 Little Temporal Aspects ............................. 25
    5.5.2 Plenty Temporal Aspects ............................ 26
  5.6 Summary ................................................ 28
## 6 Benchmark Definition

6.1 Conceptual Overview .................................................. 29
   6.1.1 Schema ............................................................ 29
   6.1.2 Data Generation .................................................. 29
   6.1.3 Queries ............................................................. 31
   6.1.4 Execution .......................................................... 31

6.2 Implementation .......................................................... 31
   6.2.1 Stock Data generation .......................................... 31
   6.2.2 Represent the Schema in the Generic Database Benchmarking Service ........... 32
   6.2.3 Import LDBC Output Into the Benchmarking Service ............................. 32
   6.2.4 History Generation .............................................. 33
   6.2.5 Generated Data Sets ........................................... 36
   6.2.6 Database Population ............................................ 37

6.3 Queries ................................................................. 37
   6.3.1 Query Definition ............................................... 37
   6.3.2 Parameters ....................................................... 37
   6.3.3 Query Execution ................................................. 40

6.4 Summary ................................................................. 40

---

## 7 Experiments

7.1 Setup ................................................................. 41

7.2 Optimization, Profiling and Query Plans .......................... 43
   7.2.1 Neo4j Query Optimization .................................... 43
   7.2.2 Neo4j Index Usage ............................................ 46
   7.2.3 System d and System m Query Plans and Indexes ................. 48
   7.2.4 What Is Measured? ............................................. 49

7.3 Measurements .......................................................... 50
   7.3.1 Queries 1.1, 1.2, 1.3, 2.2, 2.3, and 3.3 ...................... 50
   7.3.2 Queries with Graph Traversal ................................ 61

## 8 Conclusion

Appendices

A Listings for the Data Generator ...................................... 74

B Figures for the Implementation Chapter ............................ 78
   B.1 Data Distribution of the Generated Datasets ....................... 80

C Benchmark Queries .................................................... 83
Chapter 1

Introduction

1.1 Motivation

Social networks, communication network analysis, or road networks are only three examples out of the field of graphs that change in time. Graphs that change in time are graphs that receive or lose nodes or edges or have simple attribute changes over a time period. A lot of research in this field can be found that defines temporal graphs, time varying graphs or dynamic graphs in [1], [15], or [19]. They analyze how package flows in networks change over time or how and which people become important in social networks and influence others. The referenced papers define metrics to analyze the mentioned problems. Actual systems that support temporal graphs are rare. There are several graph database products available, both commercial or non-commercial, but none of these products support the time dimension natively. This is contrary to traditional relational database products which have begun to support temporal aspects in the last few years. In this thesis we try to work towards the goal that graph database vendors can start implementing temporal operations into their products.

1.2 Problem Statement

To build, compare and evaluate temporal graph databases, benchmarks must be available. There exist the TPC[4] benchmarks for relational database systems and TPC-BiH[12] benchmark for temporal relational databases. As of today, there exist no benchmarks for temporal graph databases. To implement a benchmark for temporal graph databases, general use cases must be identified and classified. Out of these use cases a benchmark shall be developed which can be used to evaluate the performance of temporal graph databases to find bottlenecks and to improve them. To verify the benchmark, it shall be run with different database systems.

1.3 Contribution

In this thesis we find and define general purpose use cases for a temporal graph database. We classify these use cases into three dimensions, whether they are data or graph centric and how they need to be combined with temporal aspects. Especially, when asking Time Range queries combined with graph traversal queries, like queries for Shortest Paths, we have to define, what such Temporal Shortest Paths look like and how graph database queries must support them. Out of these use cases, we define a benchmark consisting of database queries and a data generator.
that allows the generation of history data. The focus lies on the benchmark definition, rather than on algorithms and data structures that could improve certain temporal operators. We do the implementation of the data generator using a combination of the existing benchmarking frameworks [10] and [2]. Using these frameworks we implement database queries in the query dialects for Neo4j, IBM DB2 and SAP HANA. During the implementation we gained a lot of insight into how a temporal graph model has to be stored in a current graph database, and how queries on Neo4j need to be optimized to not run into out-of-memory or timeout errors.

1.4 Outline

Chapter 2 contains related work explaining temporal relational databases, graph databases, research in the field and benchmarks. In chapter 3 we define an examples setting, that will be used through the rest of the thesis. Chapter 4 gives an introduction to temporal graphs and how they can be stored in a database. Chapter 5 defines the use cases which lead to the benchmark definition and implementation in chapter 6. In chapter 7 three database systems are compared using the implemented benchmark and in the last chapter 8 we present our conclusion.
Chapter 2

Related Work

This chapter summarizes the related work of the thesis: First we explain the field of temporal relational databases, followed by a section about temporal graphs. At the end we will at graph databases and what benchmarks exist in the field of temporal (graph) databases.

2.1 Temporal Relational Databases

A temporal database is a database that keeps record of changes in rows, such that old values still can be accessed via SQL extensions for temporal data. When using data manipulation (DML) statements like INSERT, UPDATE and DELETE, old rows will be kept and marked with a validity interval that indicates a time interval during which a row was active. For an RDBMS this can be implemented with real world time stamps with a start and end time or with a logical time such as transaction numbers.

The research field of temporal relational databases goes back to Snodgrass et al. [18] who extended SQL92 based on the Bitemporal Conceptual Data Model. Bitemporal means the distinction of transaction time, and its orthogonal pendant valid time. Based on these ideas, Martin Kaufman wrote his PhD thesis[9] for main memory databases and indexes for temporal data. He also implemented the Timeline Index[11], for indexing time dimensions with relational databases. Kaufmann uses the synonyms system time for transaction time and application time for valid time. Using system time one keeps track of changes made to tuples. Every time a tuple is manipulated via DELETE, INSERT, UPDATE statement, it gets a new version in system time. Keeping track of real world dates like invoice dates, etc, can be done using application time. Application time stamps are driven by the corresponding use cases and may be altered or reset while system time stamps generally are set upon transaction commit and cannot be altered any more. Based on this model, Kaufmann defines several temporal operators of which we use the timeslice, also called point in time operation, and the time range operation. The former operator is used to constrain tuples to a single point in time (system or application time). The latter is used to constrain tuples to a time range.

There are several commercial databases which support temporal operations:

DB2 IBM's DB2 database has support for temporal data management. It stores data in two separate tables per entity. The first table keeps all the active rows, meaning the rows that are currently valid. The second table keeps the history of each tuple and gets updates as soon as the tuple is manipulated via DML statements. Using DB2's SQL language extensions one is able to make Point in Time and Time Range queries in both system and application time.
Oracle  Newer versions of Oracle support system time and application time using its Flashback technology. This technology is based on the archive log the database keeps. The SQL syntax allows making Point in Time, as well as Time Range queries in both time dimensions.

HANA  SAP’s HANA database is an in-memory, column-oriented database system with native temporal data management support for system time. It keeps its data in a history table having additional attributes validfrom and validto per tuple. These attributes indicate the system time range when the tuple was active. The database system keeps history data in different partitions: Currently visible tuples are kept in the current and older versions are kept in the history partition. The SQL syntax allows Point in Time queries.

2.2 Temporal Graphs

Based on the ideas of temporal databases research started around temporal graphs. [17], [1], [15] and [19] define different aspects of temporal graphs and temporal networks. Especially graph traversal and shortest path based temporal metrics are defined. Dina Yunusova summarizes many of these papers in her master thesis[20] and defines a model and a classification for temporal graph metrics. Ideas in chapter 4 are based on her master thesis.

2.3 Graph Databases

In latest history and with the trend of NoSQL databases, graph databases became popular. Wikipedia currently lists more than forty projects in the commercial and open source field. The release dates in that list indicate lots of development in the last two years. In contrast to RDBMS, where schemata with constraints are used to define entities and their relationships, graph databases use nodes, edges and attributes for that purpose and propagate a schema-less paradigm. This structure allows dynamic addition of new entities and flexible handling of relationships just by adding new attributes or edges to the graph. Graph databases distinguish themselves from RDBMS through the support of graph traversal queries which involve shortest paths and depth first searches traversing arbitrary edges and nodes.

Neo4j  For this thesis we used Neo4j[7]. Neo4j provides a graph database implemented in Java that is currently very popular, according to their customers page. It provides a web front end for development and testing purposes, showing nodes and edges as can be seen in Figure 2.1.

On one hand it has a lightweight API that allows it to be included into Java code and run as embedded database. If it shall be used as a standalone database, clients communicate via a RESTful[5] web service using an API similar to its Java API. It supports transactions and the latest development version (2.2.0) also provides a user management. There is also a project on github[2] that provides a JDBC driver wrapping a REST client for the database system. Using that driver it is easy to integrate Neo4j into already existing applications. Neo4j has its own data manipulation and query language called Cypher[3] that allows graph manipulation and graph queries similar to SQL. Opposed to the main relational databases, Neo4j does not provide any kind of temporal data management. Having the dynamic structure of nodes and edges it is easy to simulate temporal data management by adding time ranges to the edges and nodes requiring it. Chapter 4 shows how this can be accomplished.

1 http://en.wikipedia.org/wiki/Graph_database#Graph_database_projects
2 https://github.com/neo4j-contrib/neo4j-jdbc
2.4 Benchmarks

TPC and TPC-BiH The Transaction Processing Performance Council[4] provides the state-of-the-art benchmarks for relational databases for several use cases. They provide facilities to generate benchmark data and the queries to actually measure the performance of databases loaded with that data. The benchmarks are vendor independent and thus allow the comparison of databases and database features of different vendors. Kaufmann et al. provide with TPC-BiH[12] a benchmark based on the TPC-H profile to measure bitemporal databases. They also built a framework[10] which allows easy implementation of general purpose database benchmarks involving test data generation and measurements. In this thesis this framework was extended to be able to benchmark Neo4j databases having temporal properties.

LDBC - Social Network Benchmark The Linked Data Benchmark Council[3] provides with the Social Network Benchmark[2] a benchmark which is publicly available on github⁴. It is used to measure graph database performance. They provide a data generator that can be used to generate a synthetic social network, involving people that post messages in forums. The generator is parameterizable and allows scaling the output size. It also forms the basis for the generation process of the benchmark defined in this thesis. The social network benchmark also provides a bunch of queries that can be used to measure interactive workload performance. These queries were not used since they have no classification and are rather complex to be combined with temporal aspects.

⁴https://github.com/ldbc
Chapter 3

Background

Throughout the rest of the thesis we will be using examples set in the social network world which is taken from the LDBC Social Network Benchmark. This chapter briefly explains the examples setting with the entities used.

3.1 Examples Setting

We will use the entities shown in Figure 3.1. This figure shows the nodes University, Person, Post, Comment, and Forum with their corresponding attributes. A Person studies at a University which is depicted using STUDY_AT edges. Undirected KNOWS edges are representing the concept that two people know each other. HAS_CREATOR edges are used to indicate that a Person wrote the corresponding Post or Comment. Comments are always replies to an initial Post or other Comments which is shown using REPLY_OF edges. CONTAINER_OF edges are used to show which Posts and Comments belong to which Forum. A Forum always has a moderator and members which is indicated by using HAS_MODERATOR and HAS_MEMBER edges.

In the examples in the rest of the thesis it will often be the case that only specific parts of the examples setting will be used. For example only the subgraph using Person and University nodes and STUDY_AT edges might be sufficient to explain certain details and concepts. In such cases the rest of the setting will be omitted. It might also be that extra attributes will be added or omitted if it is required in particular examples.
Figure 3.1: The examples setting.
Chapter 4

Temporal Graphs

This chapter gives an overview about temporal graphs. In the first part we explain the different types of temporal graphs and use hereby Dina Yunusova’s master thesis[20] as a guideline. In the second part several approaches to saving temporal graphs in databases are discussed. For the rest of the document we will be using only one time dimension (system time), since it reduces complexity in working towards a graph database benchmark.

4.1 Graphs

A graph is commonly defined as ordered pair of vertices, also called nodes, and edges: $G = (V, E)$. $V = \{v_1, v_2...v_n\}$ is the set of all vertices and $E \subseteq \{\{u, v\} \subseteq V\}$ the set of edges. An edge is a pair of vertices and represents a link between the two vertices. Edges can be undirected or directed. In the latter case the order of the vertices in an edge matters and the graph $G$ is called digraph or directed graph. A subgraph $G_{\text{sub}} = (V_{\text{sub}}, E_{\text{sub}} : V_{\text{sub}} \subseteq V, E_{\text{sub}} \subseteq E)$ is a subset of nodes and edges from the original graph.

For our purposes every node and edge can have an attached set of attributes. These attributes are key-value pairs. As an example, in a social network graph people can be represented as nodes and friendships as edges. A person can have the attribute “eye color” that has a certain value, e.g “green”. A friendship edge could have an attached attribute “establishedAt” with value “holidays”, which means that they got to know each other during their holidays.

4.2 Temporal Graph

We define temporal graphs as graphs where each node and edge has an attached validity interval $i_{\text{val}} = [t_i, t_j]$. This interval represents the time span from $t_i$ to $t_j$ in which the node or edge with its current attributes was visible or active in the graph. The time stamps can be either logical times, like transaction numbers, or real world dates. In a temporal graph, nodes, edges and attributes can emerge, disappear or be changed, which happens as follows:

- **Node/edge is created:** When a node or edge with its initial attributes gets created, its validity interval is set to $i_{\text{val}} = ['now', \infty)$. This means that the newly created node or edge is valid from now on and it keeps being currently valid.

- **Node/edge is deleted:** When a node or edge shall be removed from the current view of a graph, its validity interval is set to $i_{\text{val}} = ['creationDate', 'now')$. This means that the
node or edge was valid in the graph from \textit{creationDate} up to \textit{now}.

- **Node/edge or attribute gets changed:** In this case the following procedure happens:

  1. Set validity interval of the node/edge to $i_{\text{val}} = ['creationDate', 'now')$. This ensures that we know when the old values were valid.
  2. Copy the node/edge with all its attributes to a new node or edge.
  3. Add/update/delete attribute value in the new node/edge.
  4. Set validity interval of new node/edge to $i_{\text{val}} = ['now', \infty)$. This ensures that the new values are the ones currently active.

This procedure guarantees that a history over nodes and edges is kept inside the temporal graph when changes happen. It also assumes the \textit{spatial referential integrity} property, which is described in section 4.3.1.

The definition of the described model is slightly simpler than the one explained in [20]. Yunusova additionally attached attributes to a graph and more important, she attached validity intervals to each single attribute. With this approach one could model a more fine grained semantics where edges and nodes do not change their validity intervals upon changes while individual attributes do. In terms of simplicity and considering implementing a graph benchmark that analyses graph metrics like shortest paths or node centrality, we will stick to the simpler model described above. This model can still be extended in future work to improve the benchmark quality.

As an example, Figure 4.1 shows a temporal graph of a social network: People study at universities over a period of time $T_1 \ldots \infty$. In this figure, the node Sandra has a validity interval of $[T_2, \infty)$ and the STUDY_AT edge from Sandra to ETH Zürich has a validity interval of $[T_2, T_5)$.

In [20, p. 118], Yunusova classifies three different types of temporal graphs, which we will reuse in the following subsections.

### 4.2.1 Temporal Subgraph

A \textit{temporal subgraph} is a subset of nodes and edges of a temporal graph where all the nodes and edges are valid in a specified time range. In other words, to generate a temporal subgraph from a temporal graph, one has to specify a time range $i_{\text{range}} = [t_i, t_j)$ and the generator picks all the nodes and edges from the temporal graph that have overlapping validity intervals with $i_{\text{range}}$.

Having a temporal graph or subgraph allows us to create queries over several dimensions:

1. **Queries Over Attributes:** Analogous to database queries, one can create queries that ask for nodes/edges having specific attributes (SQL \texttt{WHERE}), join attributes (SQL \texttt{JOIN}) and aggregate attribute values (SQL \texttt{COUNT()}, \texttt{AVG()}, \texttt{MAX()}, \texttt{MIN()}, \texttt{GROUP BY}). The join is a special case in graphs as we consider a join as a search for existing edges between two node types. For example in the subgraph of \texttt{Forums} and \texttt{Persons}, nodes can be joined if there exist \texttt{HAS_MEMBER} edges between them.

2. **Graph Related Queries / Graph Metrics:** On a graph one can create queries that require graph traversal, which cannot simply be done in classic RDBMs. This quickly goes into the field of graph computation, so we will only consider rather simple metrics based on shortest paths and node distances.
3. **Time Related Queries**: Here we can distinguish between *time related queries combined with queries over attributes* and *time related queries combined with graph related queries*. The first combination either finds nodes/edges or aggregates given some criteria in the given time range while the second combination calculates temporal versions of graph metrics that can be found in literature (temporal shortest paths, temporal network diameters, etc)[19],[20].

Section 5.1 will elaborate more on these types of queries and how they can be combined.

### 4.2.2 Fixed Temporal Graph

A *Fixed temporal graph* is a subset of nodes and edges of a temporal graph where all the nodes and edges are valid at a specified *point in time*. In other words, when generating a fixed temporal graph, the generator picks all the nodes and edges that contain the specified point in time in their validity interval. In literature [20] one often also encounters the terms *timeslice*, *snapshot* or *time travel* for the same type of graph generation or result computation.

Queries on fixed temporal graphs can have the same dimensions like temporal subgraphs:

1. **Queries Over Attributes**: Analogous to subsection 4.2.1.
2. **Graph Related Queries / Graph Metrics**: Analogous to subsection 4.2.1.
3. **Time Related Queries**: Analogous to subsection 4.2.1 we distinguish between time related queries combined with queries over attributes and time related queries combined with graph related queries. The combinations are now either nodes/edges or aggregates given some criteria at the given point in time or non temporal graph metrics at a given point in time. E.g the shortest paths or the network diameter at a given point in time.

### 4.2.3 Static Graph

Coming from a fixed temporal graph one can define a static graph as a projection of all nodes and edges and all their attributes but without validity intervals. The result is basically a graph as defined in section 4.1 that has no knowledge about any temporal aspects. For the sake of completeness queries can have the following dimensions on a static graph:

1. **Queries Over Attributes**: Analogous to subsection 4.2.1.

2. **Graph Related Queries / Graph Metrics**: Analogous to subsection 4.2.1.

### 4.3 Storing a Temporal Graph

Since we want to work towards a graph database benchmark, we keep in mind the way the data has to be saved in the database. In the next subsection we will explain the ways this can be done. The first approach will save the nodes and edges of temporal graphs in tables of temporal relational databases. The second approach will use a graph database and validity intervals per node and edge to store the temporal graph. For the examples, we will use the temporal graph from Figure 4.2: At $T_1$ a Person node is created that gets company of a University node at $T_3$. The Person starts to study at the University at $T_4$ in the bachelor’s program and changes its email address at $T_6$. At $T_{10}$ the Person switches to the master’s program.

#### 4.3.1 Temporal Relational Database

If the temporal graph shall be stored in an RDBMS, we have several ways to accomplish this task. In this section, we assume that the RDBMS has no temporal support at all and we have to keep track of validity intervals on our own. Each of the possible methods has advantages and disadvantages and two of them are described in the next subsections.

**Dynamic Nodes and Edges**

Figure 4.3 shows a dynamic approach with four tables to store the temporal graph. With this approach one can add different types of nodes at runtime. This means we can add for example a node of type Post with any corresponding attributes without changing the layout of the tables. This technique also allows keeping track of time intervals for individual attributes, which is close to the model described in [20]. One drawback of this method is that several table joins are needed to get all data of only one node or edge, which potentially leads to performance issues. The design also does not respect different data types, for example dates or numbers. This can be tackled with attribute tables per data type which in turn leads to even more joins per node or edge. Data consistency constraints need to be implemented to keep validity intervals of attributes inside the validity intervals of their corresponding nodes and edges. Constraints for edges also need to be set, so that their validity intervals remain inside the ones of the nodes an edge belongs to.
Figure 4.2: A temporal graph.
4.3. STORING A TEMPORAL GRAPH

Figure 4.3: A temporal graph saved in an RDBMS with dynamic nodes and edges.

Static Node and Edge Types

Another approach is shown in Figure 4.4. Every node type and every edge type gets its own table with its corresponding attributes. Every table has its own validity interval which states in which period a tuple was active. In contrast to the dynamic approach this method has the advantage that fewer joins are needed to get all data of nodes and edges. The RDBMS allows defining data types and constraints per attribute which is another benefit of this method. The simpler design also requires only constraints that ensure that validity intervals of edges remain inside the validity intervals of their corresponding nodes. With this approach, we also cannot keep track of individual attribute changes and have to replicate node rows if such an attribute changes. This approach is used in chapter 6 and chapter 7 to store data in relational databases.

Spatial Referential Integrity

Because we have an RDBMS and we keep the nodes and edges saved in either way described above, we inherently get a property we call spatial referential integrity. It means that an edge possibly belongs to more than one row if a node has history and thus is saved on several rows, differing in attributes and validity intervals. This is shown in Figure 4.4: The first STUDY_AT edge is present between the first row of Person and the first row of University but also between the second row of Person and the first row of University. This is because we keep referential integrity only on the id columns but not on the temporal columns. In an RDBMS this seems to be a logical way to implement such temporal aspects but when we switch to graph databases, we need to have this concept in mind to implement the model in a good way.

4.3.2 Graph Databases

Today’s graph databases support saving nodes and edges with attributes. The attributes are given as key value pairs and have a data type. Since there is no inherent support for temporal aspects, this has to be modeled using nodes and edges. In the following subsections, we will show two ways how a temporal graph can be stored in a graph database.
Naive Approach

In a first approach we add a validity interval to each node and each edge. Whenever an attribute of a node changes, the node gets copied and the validity interval of the old node stops now and the validity interval of the new node starts from now. The same applies to edges. If we now have copies of nodes we need to make sure that the structure of the graph remain intact: Edges need to be reestablished between original and copied nodes and their original neighbors. To achieve that we replicate all the edges of a node upon copying it. During the replication we set the validity intervals of the old edges to end now and the new edges start their validity intervals now. Figure 4.5 shows our example temporal graph in this form. This approach has the huge drawback of possibly replicating the whole graph: Assume the temporal graph is a dense graph that is, the number of edges close to the number of possible edges, or a star with a central node. If the central node is updated often and we keep the history the way we explained here, we have to replicate all the edges as well. This is a huge effort that brings no benefit for querying the graph, since queries have to crawl even more edges than necessary. If we take a look at the edges, we see that with such an approach we have spatial referential integrity plus referential integrity in the time dimension: Edges belong to exactly one node pair and are copied with split time intervals if time properties of nodes change. But spatial referential integrity should suffice for our purposes. To overcome this shortcoming we use the method described in the next subsection.

Separate Structure from State

To achieve spatial referential integrity only, we use the method described in a blog post by Ian Robinson [16]. The author separates structure (edges) from state (attributes of nodes). One original node is split up into at least two different nodes, an identity node and state node containing the original node’s attributes. These nodes are connected by state edges. A state edge contains the validity interval that indicates in what period the original node had the corresponding state. When the original node’s attributes are changed, a new state node is created and connected to the identity node via a new state edge, having the new validity interval. The structure of the original temporal graph is kept inside structural edges between the identity nodes. The benefit of
this approach is that it has spatial referential integrity: It allows changing attributes without influencing the structure of the original temporal graph and therefore introducing additional edges. Using this approach, one has to introduce data integrity constraints on the validity intervals of the edges so that state edges to the same identity nodes do not overlap and are consecutive. Depending on the application, it may be important to introduce similar constraints on the structure edges, if these validity intervals must not overlap. Figure 4.6 shows what this looks like in our example temporal graph: The Person is split up into three different nodes, an identity node Person and two state nodes. The University node is split up into one state node and one identity node University. The original STUDY_AT edge is represented with two STUDY_AT edges with non overlapping validity intervals. This approach will be used in chapter 5 and chapter 6 to save the temporal graph model in a graph database.
CHAPTER 4. TEMPORAL GRAPHS

Figure 4.6: A temporal graph saved in a graph database with separation of structure and state.
Chapter 5

Use Cases

This chapter identifies general use cases of temporal graph databases. It is influenced by use cases identifiable on relational databases but extends to use cases that can not be expressed with relational databases. By the term “use case”, we mean database queries that can be issued on temporal graph databases. In the first part of the chapter we classify types of queries for which in the second part example queries are formulated. In the following chapters the example queries will be reused to implement the actual benchmark.

5.1 Query Types

As already mentioned in chapter 4 there are different types of queries that can be classified on a temporal graph. In this section we look at three dimensions in more detail such that we can define concrete queries in the following sections.

5.1.1 Queries Over Attributes

The class Queries Over Attributes contains Attribute, Join and Aggregate:

- A simple Attribute query asks for nodes that have an attribute set to a given value.
- A Join asks for pairs of nodes having a specified edge.
- An Aggregate is a computed value over a set of attributes of specified nodes.

5.1.2 Graph Related Queries / Graph Metrics

The class Graph Related Queries / Graph Metrics contains the bunch of queries that use graph traversal. We will use Shortest path, Closeness Centrality and Network Diameter. These graph related queries get increasingly more complex: While Shortest Path presumably has to touch a smaller part of the graph, Closeness Centrality has to touch more and with Network Diameter even the whole graph has to be crawled, possibly several times. With regard to building a benchmark it also has to be mentioned, that these three metrics can easily be expressed with a query language of a graph database.

- The Shortest Path is calculated between exactly two given nodes. It is defined as a path that connects two nodes with at as few edges as possible. The number of edges on the shortest path is called the distance $d_{ij}$ between the nodes $i$ and $j$. 
• The *Closeness Centrality* is an indicator for how long it will take to spread information from a node \( i \) to all other nodes in a network. It is calculated between one given node and all its connected nodes. [15] define the closeness centrality as the inverse of the average distance from node \( i \) to any other node in the network:

\[
C_i = \frac{N - 1}{\sum_j d_{ij}}
\]

where \( N \) is the number of nodes in the graph. As already mentioned, we will use this metric because it is relatively easy to compute but has more complexity than the *Shortest Path* metric.

• The *Network Diameter* is calculated over all pairs of nodes. [20] describes the network diameter as the length of the longest shortest path in a graph. This metric is also easy to compute but has even more complexity than *Closeness Centrality*.

### 5.1.3 Time Related Queries

*Time Related Queries* are orthogonal to the ones above and thus can be combined with them. We mainly distinguish the following query types:

• The *Time Range* query is used to constrain the graph data to a certain time range \( t_{range} = [t_i, t_j) \). It will generate a temporal subgraph as explained in subsection 4.2.1.

• The *Point in Time* query constrains a graph to all nodes and edges visible at a certain point in time. It will generate a fixed temporal graph as explained subsection 4.2.2.

• The *Non Temporal* query can be seen as *Time Range* query over the whole time dimension where in practice the temporal aspect does not matter any more. It will produce a static graph as explained in subsection 4.2.3.

Because *Time Related Queries* are orthogonal to *Graph Related Queries / Graph Metrics* and *Queries Over Attributes* we can use them as a starting point for defining concrete example queries on (temporal) graphs. This is done in the next sections: Each of the *Time Related Queries* types is used as a basis that is combined with the other two query types.

### 5.2 Examples Setting

As examples setting we use the data model shown in chapter 3. In the queries for fixed temporal graphs and temporal subgraphs we assume that each node and edge type (*Person*, *University*, *Forum*, *Post*, *Comment*, *KNOWS*, *STUDY_AT*, etc.) has attached a validity interval as described in subsection 4.2.1.

### 5.3 Non Temporal Queries - Queries on a Static Graph

Static graphs represent *Non Temporal Queries* as defined above. On a static graph *Queries Over Attributes* and *Graph Related Queries / Graph Metrics* can be combined. Table 5.1 shows the pairwise combinations of these query types. The cells on the diagonal are no combinations, but simple queries.
The following list explains the combinations from Table 5.1 using concrete queries inside the examples setting. These queries will later on be used to implement the benchmark for different database systems. There are no combination of Aggregate with Closeness Centrality or Network diameter because the latter two are already some type of aggregate over the whole graph using graph traversal mechanisms. There is also no reasonable combination of Shortest Path with either Closeness Centrality or Network Diameter.

1.1 Attribute

a. Query all Person nodes with a given date of birth.
b. Query all women/men.
c. Query all Comments and Posts written in a given language.
d. Query all Person nodes with date of birth > 'yyyyymmdd'.
e. Query all Posts with a length > X.

1.2 Attribute – Join

a. Query all University nodes with students having a given date of birth.
b. Query all University nodes with female/male students.
c. Query all Person nodes that are CREATOR_OF Posts in a given language.
d. Query all Person nodes that are CREATOR_OF Comments and Posts with length > X.

1.3 Attribute – Aggregate: Aggregates over values having other attributes constrained.

a. Count all men/women.
b. Count Comments and Posts grouped by browsers where length > X.
c. Count Comments and Posts grouped by length where browser is Y.

1.4 Attribute – Shortest Path: Shortest Path with attributes constrained.

a. Calculate the shortest path between two people A and B in the Person/KNOWS subgraph using only edges to men/women.

1.5 Attribute – Closeness Centrality: Closeness Centrality having attributes constrained.

a. Get the closeness centrality of a Person within men/women in the Person/KNOWS subgraph.
CHAPTER 5. USE CASES

b Get the closeness centrality of a Person within the Person/KNOWS subgraph using browser Y.

1.6 Attribute – Network Diameter: Network Diameter of a subgraph having attribute constrained.
   a Get the network diameter of women/men in the Person/KNOWS subgraph.

2.2 Join
   a Query all University nodes with their students.
   b Query all Person nodes with their Comments and Posts.

2.3 Join – Aggregate:
   a Get the number of students grouped by university.

2.4 Join – Shortest Path:
   a Get the shortest path between two people in the Person/STUDY_AT/University subgraph using only STUDY_AT edges.
   b Get the shortest path between two people in the Forum/HAS_MEMBER/Person subgraph using only HAS_MEMBER edges.

2.5 Join – Closeness Centrality
   a Get the closeness centrality of a person in the Person/STUDY_AT/University subgraph using only STUDY_AT edges.
   b Get the closeness centrality of a person in the Forum/HAS_MEMBER/Person subgraph using only HAS_MEMBER edges.

2.6 Join – Network Diameter
   a Get the network diameter of the Person/STUDY_AT/University subgraph using only STUDY_AT edges.
   b Get the network diameter of the Forum/HAS_MEMBER/Person subgraph using only HAS_MEMBER edges.

3.3 Aggregate
   a Count all Comments and Posts grouped by browsers used.
   b Count all Person nodes grouped by browsers used.
   c Count all Comments and Posts grouped by length.

3.4 Aggregate – Shortest Path: Aggregate attributes along the shortest path.
   a Get the length of the shortest path in the Person/KNOWS subgraph.
   b Get the sum of all lengths of Comments and Posts in the Comment/Post/REPLY_OF subgraph between a given Comment and an initial Post the Comment is eventually replying to.

4.4 Shortest Path
5.4 Point in Time Queries - Queries on a Fixed Temporal Graph

As defined in section 5.1 fixed temporal graphs can be generated with Point in Time queries. Such Point in Time queries are now easily combined with all the queries from the list defined in section 5.3, if we extend each of them with a “as of point in time $D$” clause. We do not replicate the whole list here but show only an example of the first two queries:

1.1 Attribute:
   a. Query all Person nodes with a given date of birth as of point in time $D$ – be aware that the date of birth has nothing to do with $D$.
   b. Query all women/ men as of point in time $D$.

5.5 Time Range Queries - Queries on a Temporal Subgraph

As defined in section 5.1, temporal subgraphs can be generated with Time Range queries. It is not as easy to extend all the queries from the list in section 5.3 with an “in a given time range” clause as was done in the previous section. The combination can be done with the query types Attribute, Join and Aggregate which we call queries with little temporal aspects. To make the combination of Time Range queries with Shortest Path, Closeness Centrality and Network Diameter we need to define how such metrics would look like in a temporal range, which is done in subsection 5.5.2.

5.5.1 Little Temporal Aspects

The queries in this subsection do not consider graph traversal aspects that have to be combined with a time range, but only ask for values on nodes/edges in a given time range. Table 5.2 shows the combinations of query types with little temporal aspects which can be combined with an “in a given time range” clause.

<table>
<thead>
<tr>
<th></th>
<th>Attribute</th>
<th>Join</th>
<th>Aggregate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
<td>1.1</td>
<td>1.2</td>
<td>1.3</td>
</tr>
<tr>
<td>Join</td>
<td>-</td>
<td>2.2</td>
<td>2.3</td>
</tr>
<tr>
<td>Aggregate</td>
<td>-</td>
<td>-</td>
<td>3.3</td>
</tr>
</tbody>
</table>

Table 5.2: Combinations of query types on temporal subgraphs with little temporal aspects.
The diagonal from Table 5.2 represents the following types of queries:

1.1 Attribute: Ask for nodes or edges which have a given attribute and lie in the given time range.

2.2 Join: Find pairs of nodes having a specified edge which all lie in the given time range.

3.3 Aggregate: Aggregate values of attributes over sets of nodes or edges lying in the given time range.

Like we did in section 5.3, the query types Attribute, Join and Aggregate can also be pairwise combined, which yields the combinations 1.2, 1.3, 2.3. To get the concrete queries we can now simply add “in a given time range” to the queries 1.1, 2.2, 3.3, 1.2, 1.3 and 2.3 from section 5.3. As this is easily imaginable, we will not replicate the list here.

5.5.2 Plenty Temporal Aspects

As already mentioned, there is more work to do if we want to combine Time Range queries with Graph Related Queries / Graph Metrics. If we want to have temporal versions of Shortest Path, Closeness Centrality and Network Diameter, we need to define some terms previously:

- **Temporal Walk**: [15] defines a temporal walk from node $i$ to node $j$ as a sequence of $L$ edges $[(n_0, n_1), (n_1, n_2), \ldots, (n_{L-1}, n_L)]$ and an increasing sequence of times $t_1 < t_2 < \ldots < t_L$, such that there is an edge in the temporal graph for each $t_i \in t_1 \ldots t_L$. We lessen the constraint to $t_1 \leq t_2 \leq \ldots \leq t_L$, which means, there may be subsequent edges having the same or overlapping time stamps.

- **Temporal Path**: [15] describes a temporal path as a temporal walk without repeated nodes.

- **Temporal Length**: [15] describes temporal length, or duration, as the time interval between the first and the last contact in the temporal path.

We can now define the three metrics used for the queries:

- **Temporal Shortest Path**: Since there are several definitions for a temporal shortest path (earliest-arrival path [19], latest-departure path[19], etc) we are going to use the fastest path [19], which is defined as the temporal path from node $i$ to node $j$ with the minimal temporal length. Using the temporal shortest path, we use [15] to define the temporal node distance $\hat{d}_{ij}$ (also the fastest distance [17]): The temporal node distance from node $i$ to node $j$ is the temporal length of the Temporal Shortest Path from $i$ to $j$. The decision to use the fastest path is based on the fact that the literature provides definitions of Temporal Closeness Centrality and Temporal Network Diameter which base on it.

- **Temporal Closeness Centrality**: [15] defines this metric the same way as the static property but now over temporal distances:

$$C_t = \frac{N - 1}{\sum_{j} \hat{d}_{ij}}$$

where $\hat{d}_{ij}$ is the temporal node distance between $i$ and $j$. The main reason to use the Temporal Closeness Centrality is to have similar metric to the one on static graphs which is more complex than Temporal Shortest Path.
5.5. **TIME RANGE QUERIES - QUERIES ON A TEMPORAL SUBGRAPH**

- **Temporal Network Diameter:** [15] extends the static graph property *Network Diameter* to be the largest temporal distance between any pair of nodes

\[
D = \max_{i,j} \hat{d}_{ij}
\]

As *Network Diameter* on static graphs does, *Temporal Network Diameter* introduces a higher complexity than *Temporal Closeness Centrality*.

Using these definitions, it is possible to combine the query types *Attribute*, *Join* and *Aggregate* with *Temporal Shortest Path*, *Temporal Closeness Centrality* and *Temporal Network Diameter* as shown in Table 5.3.

<table>
<thead>
<tr>
<th></th>
<th>Temporal Shortest Path</th>
<th>Temporal Closeness Centrality</th>
<th>Temporal Network Diameter</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Attribute</strong></td>
<td>1.4</td>
<td>1.5</td>
<td>1.6</td>
</tr>
<tr>
<td><strong>Join</strong></td>
<td>2.4</td>
<td>2.5</td>
<td>2.6</td>
</tr>
<tr>
<td><strong>Aggregate</strong></td>
<td>3.4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Temporal Shortest Path</strong></td>
<td>4.4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Temporal Closeness Centrality</strong></td>
<td>-</td>
<td>5.5</td>
<td>-</td>
</tr>
<tr>
<td><strong>Temporal Network Diameter</strong></td>
<td>-</td>
<td>-</td>
<td>6.6</td>
</tr>
</tbody>
</table>

Table 5.3: Combinations of query types on temporal subgraphs with plenty temporal aspects.

Using this table, the following queries can be defined on our examples setting:

1.4 **Attribute – Temporal Shortest Path**
   a. Calculate the temporal shortest path between two people \(A\) and \(B\) in the **Person/KNOWS** subgraph using only edges between men/women.

1.5 **Attribute – Temporal Closeness Centrality**
   a. Get the temporal closeness centrality of a **Person** within men / women in the **Person/KNOWS** subgraph.
   b. Get the temporal closeness centrality of a **Person** within the **Person/KNOWS** subgraph of people using browser \(Y\).

1.6 **Attribute – Temporal Network Diameter**
   a. Get the temporal network diameter of men in the **Person/KNOWS** subgraph.

2.4 **Join – Temporal Shortest Path**
   a. Get the temporal shortest path between two people in the **Person/STUDY_AT/University** subgraph using only **STUDY_AT** edges.
   b. Get the temporal shortest path between two people in the **Forum/HAS_MEMBER/Person** subgraph using only **HAS_MEMBER** edges.

2.5 **Join – Temporal Closeness Centrality**
   a. Get the temporal closeness centrality of a **Person** in the **Person/STUDY_AT/University** subgraph using only **STUDY_AT** edges.
2.6 Join – Temporal Network Diameter

a Get the temporal network diameter of the Person/STUDY_AT/University subgraph using only STUDY_AT edges.

b Get the temporal network diameter of the Forum/HAS_MEMBER/Person subgraph using only HAS_MEMBER edges.

3.4 Aggregate – Temporal Shortest Path: Aggregate attributes along the temporal shortest path.

a Get the length of the temporal shortest path in the Person/KNOWS subgraph. The length is the number of nodes along the temporal shortest path.

b Get the sum of all lengths of Comments and Posts in the Comment/Post/REPLY_OF subgraph between a given Comment and an initial Post the Comment is eventually replying to.

4.4 Temporal Shortest Path:

a Get the temporal shortest path between two given Person nodes using KNOWS edges.

5.5 Temporal Closeness Centrality

a Calculate the temporal closeness centrality of a given Person node in the Person/KNOWS subgraph.

6.6 Temporal Network Diameter

a Calculate the temporal network diameter of the Person/KNOWS subgraph.

5.6 Summary

In this chapter we identified the three dimensions Queries Over Attributes, Graph Related Queries / Graph Metrics and, orthogonal to the others, Time Related Queries. In each of the dimensions we classified three types of queries. For each of the query types in Time Related Queries we combined the query types from the other two dimensions. This was easy for Static Graphs and Point in Time queries. For Time Range queries we made the distinction of little temporal aspects and plenty temporal aspects. The latter led to the definition of a bunch of new temporal graph metrics. After we did all the classifications and combinations we found concrete queries in our examples setting that will be used to implement a benchmark for temporal graph databases.
Chapter 6

Benchmark Definition

This chapter contains the definition of the benchmark. As a basis, we took the LDBC social network benchmark, because it allows generating a reasonable amount of data and is scalable in terms of output size. On top of the generated data we use the *generic database benchmarking service* [10] to generate the history. The mentioned service is also used to execute the queries, and thus perform the actual measurements explained in the subsequent chapter. The queries were executed on three different database systems: Neo4j which is a graph database, system d which is a commercial disk-based row store and system m which is a commercial in-memory column store database. We do not reveal the names of systems d and m because of license regulations.

6.1 Conceptual Overview

6.1.1 Schema

The LDBC social network benchmark uses the same entities as introduced in chapter 3. Figure B.1 shows the UML class diagram from the LDBC social network benchmark. It will be used as a basis for the temporal benchmark and was adapted to support storage of temporal aspects and temporal queries. We extended the nodes Person, Comment and Post with two attributes validFrom and validTo. The edge types KNOWS, STUDY_AT and HAS_MEMBER were altered to contain the same two attributes.

6.1.2 Data Generation

To run a benchmark, data has to be generated on which execution times of queries can be measured. Figure 6.1 shows the generation process used for our benchmark: In a first step, we used the LDBC social network benchmark to generate stock data. In a second step this data is extended with history updates and in the final step the data is imported into the database systems. While the *generic database benchmarking service* already supports populating system d and system m databases, we had to write a script for importing the generated comma separated value (CSV) files into Neo4j. For the sake of simplicity this step was not integrated into the benchmarking service.
CHAPTER 6. BENCHMARK DEFINITION

Figure 6.1: Data generation workflow.
6.2. IMPLEMENTATION

6.1.3 Queries

We translated the queries from section 5.1 into Neo4j’s Cypher and system d’s and system m’s SQL dialects. These queries were then entered into the generic database benchmarking service via its web interface.

6.1.4 Execution

The generic database benchmarking service measures execution times of queries using JDBC. The developers already provide query measurement drivers for system d and system m. Neo4j was not yet supported by the framework. To be able to run Cypher queries using the generic database benchmarking service, we integrated the Neo4j JDBC driver\(^1\) and wrote a Neo4j query driver. Using these facilities the framework can run the parametrized queries for all three different systems.

6.2 Implementation

This section will give a more detailed overview over each of the aforementioned steps.

6.2.1 Stock Data generation

To generate stock data, we use the LDBC social network data generator. This tool uses Apache Hadoop 1.2\(^2\) to generate CSV files which then can be imported in databases. The amount of generated data can be parametrized with the accompanying “params.ini” file. Listing 6.1 and Listing 6.2 show examples of the “params.ini” files which were used for the next steps to generate graph data:

Listing 6.1: params.ini for small data set

```
numPersons:100
numYears:5
startYear:2010
compressed:false
serializer:csv_merge_foreign
numThreads:1
```

Listing 6.1 tells the generator to create 100 people with all the accompanying data. The resulting CSVs have a size of about 6MB. Such a dataset is easy to use for testing the history generation process and suffices for the more complex traversal queries explained later on.

Listing 6.2: params.ini for large data set

```
scaleFactor:1
compressed:false
serializer:csv_merge_foreign
numThreads:1
```

\(^1\)https://github.com/neo4j-contrib/neo4j-jdbc

\(^2\)http://hadoop.apache.org
6.2.2 Represent the Schema in the Generic Database Benchmarking Service

In the next step, the database schema (Figure B.1) needed to be added to the generic database benchmarking service. This service allows the definition of tables with their columns and constraints. To be able to run temporal data generation, the necessary tables need to be marked as “temporal” and the time dimension columns need to be defined. In our case, the necessary temporal tables are: COMMENT, PERSON, PERSON_KNOWS_PERSON, PERSON_STUDY_AT_ORGANISATION and FORUM_HASMEMBER_PERSON. For all those tables, we added a system time dimension to the column definitions. Figure 6.2 gives a glimpse of the PERSON table in the web interface of the benchmarking service.

6.2.3 Import LDBC Output Into the Benchmarking Service

We adapted the generic database benchmarking service to process input from the LDBC social network benchmark. To this end, we implemented the new Generator classes $\text{GraphInitialGenerator}$. It is merely used to import the output of the LDBC social network benchmark into the benchmarking service’s internal representation and to enhance the entities (Person, Comment, Post, KNOWS, STUDY_AT and HAS_MEMBEER) with the temporal attributes validFrom and validTo. Figure 6.3 shows the main classes implemented for that.
6.2. IMPLEMENTATION

6.2.4 History Generation

To actually generate history data, also known as versions, we implemented *GraphHistoryGenerator*. It generates a history based on scenarios we describe in the following paragraphs. *GraphHistoryGenerator* takes a scaling factor $F$ which is configurable via the benchmarking service’s web interface. It performs $M \times F$ iterations where $M$ is a scaling multiplier. In each iteration, it executes one of the following scenarios uniformly at random with a probability $p$, and increments a variable CURRENT_TIME. The probabilities and scaling multiplier are configurable in a config file. Figure 6.4 shows the main classes implemented for this.

Figure 6.4: History Generator Classes
The next two paragraphs explain these scenarios in more detail:

Syntax  The syntax used in the scenarios is close to Neo4j’s Cypher:

- **Nodes** are written in parenthesis. First comes the variable name, second comes the label. A label can be seen similar to a type in a programming language. The label can be omitted if it is clear from context or irrelevant. Example: `(p:Person)`

- **Edges** are written in brackets, with the variable name first and the label of the edge second. To the left and right of the edge are always two nodes. `<-` and `->` show the directions of an edge if it is directed. `-` can be used if an edge is undirected. Example: `(p:Person)-[e:STUDY_AT]->(u:University)` for a directed STUDY_AT edge `e` between Person `p` and University `u`. Example: `(p1:Person)-[k:KNOWS]-(p2:Person)` for an undirected KNOWS edge `k` between Person nodes `p1` and `p2`.

**Scenarios** We defined the scenarios with the corresponding probabilities `p` shown in Table 6.1. The scenario in Listing 6.4 with its helper function defined in Listing 6.3 can be read as an example on the next page. All the other listings can be found in Appendix A.

<table>
<thead>
<tr>
<th>Scenario Name</th>
<th>Probability</th>
<th>Listing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create Comment</td>
<td>p=0.1</td>
<td>Listing 6.4</td>
</tr>
<tr>
<td>Delete Message</td>
<td>p=0.04</td>
<td>Listing A.2</td>
</tr>
<tr>
<td>Update Comment</td>
<td>p=0.1</td>
<td>Listing A.3</td>
</tr>
<tr>
<td>Delete Person</td>
<td>p=0.0005</td>
<td>Listing A.4</td>
</tr>
<tr>
<td>Update Person</td>
<td>p=0.1</td>
<td>Listing A.5</td>
</tr>
<tr>
<td>Become Friends</td>
<td>p=0.3295</td>
<td>Listing A.6</td>
</tr>
<tr>
<td>Quit Friendship</td>
<td>p=0.05</td>
<td>Listing A.7</td>
</tr>
<tr>
<td>Start Studying</td>
<td>p=0.15</td>
<td>Listing A.8</td>
</tr>
<tr>
<td>Stop Studying</td>
<td>p=0.05</td>
<td>Listing A.9</td>
</tr>
<tr>
<td>Become Forum Member</td>
<td>p=0.05</td>
<td>Listing A.10</td>
</tr>
<tr>
<td>Leave Forum</td>
<td>p=0.03</td>
<td>Listing A.11</td>
</tr>
</tbody>
</table>

Table 6.1: Defined scenarios with corresponding probabilities and listings.

The probabilities were generally set with the focus on creating friendships and comments. It also involved some tweaking to prevent nasty situations. It could for example happen that there were not enough people created such that no more friendships could be established. On the other hand, too many people could be created such that the friendship graph was only loosely connected and graph traversal queries made no more sense. The current setup represents a mix which can be used well for the benchmark with a reasonable data set size.
6.2. IMPLEMENTATION

Listing 6.3: Helper function createPerson.

```java
createPerson():
    create new (a:Person)
    set attributes at random,
    set a.validFrom=CURRENT_TIME,
    set a.validTo=∞
    return a
```

Listing 6.4: Scenario Create Comment.

```java
createComment():
1. Create new (m:Comment),
   set all attributes on m,
   set m.validFrom=CURRENT_TIME
   set m.validTo=∞
2. Uniform distribution of one of the following cases
   a) [probability=0.8]:
      uniformly at random select (a:Comment) where a.validTo=∞
      create edge (m)<-[e:REPLY_OF_COMMENT]-(a)
   b) [probability=0.2]:
      uniformly at random select (a:Post) where a.validTo=∞
      create edge (m)<-[e:REPLY_OF_POST]-(a)
3. Uniform distribution of one of the following cases
   a) [probability=0.999]:
      uniformly at random select (a:Person) where a.validTo=∞
      and edges (a)<-[e:HAS_MEMBER]-[f:Forum]-[:CONTAINER_OF]>(r) exist
   b) [probability=0.001]:
      a = createPerson()
      create edge (a)<-[e:HAS_MEMBER]-(f)
      set e.validFrom=CURRENT_TIME,
      set e.validTo=∞
4. create edge (m)<-[e:HAS_CREATOR]-(a),
   set a.validFrom=CURRENT_TIME,
   set a.validTo=∞
5. uniformly at random select (c:Country)
6. create edge (m)<-[e:IS_LOCATED_IN]>(c)
```
6.2.5 Generated Data Sets

Table 6.2 shows the final 11 generated datasets with their corresponding sizes as CSV files on disk. It represents possible combinations of the LDBC social network benchmark (100 people, 11000 people) and the generic database benchmark service (0 - 50000000 history updates).

<table>
<thead>
<tr>
<th>Generated Data Sets</th>
<th>Dataset Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 people</td>
<td>A</td>
<td>5.9 MB</td>
</tr>
<tr>
<td>11000 people</td>
<td>B</td>
<td>874.1 MB</td>
</tr>
<tr>
<td>0 history updates</td>
<td>C</td>
<td>6.9 MB</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>875.1 MB</td>
</tr>
<tr>
<td>10000 history updates</td>
<td>E</td>
<td>15.8 MB</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>883.5 MB</td>
</tr>
<tr>
<td>100000 history updates</td>
<td>G</td>
<td>109.8 MB</td>
</tr>
<tr>
<td></td>
<td>H</td>
<td>969.1 MB</td>
</tr>
<tr>
<td>1000000 history updates</td>
<td>I</td>
<td>1.1 GB</td>
</tr>
<tr>
<td></td>
<td>J</td>
<td>1.9 GB</td>
</tr>
<tr>
<td>50000000 history updates</td>
<td>K</td>
<td>5.9 GB</td>
</tr>
</tbody>
</table>

Table 6.2: Generated datasets.

Table 6.3 shows the numbers of entries per node type or edge type for the generated datasets C and J.

<table>
<thead>
<tr>
<th>Node or edge type</th>
<th>Count in dataset C</th>
<th>Count in dataset J</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person</td>
<td>107</td>
<td>625429</td>
</tr>
<tr>
<td>Comment</td>
<td>7610</td>
<td>4342556</td>
</tr>
<tr>
<td>Post</td>
<td>14156</td>
<td>1214766</td>
</tr>
<tr>
<td>University</td>
<td>6421</td>
<td>6421</td>
</tr>
<tr>
<td>Forum</td>
<td>1464</td>
<td>110202</td>
</tr>
<tr>
<td>KNOWS</td>
<td>6154</td>
<td>7005190</td>
</tr>
<tr>
<td>STUDY_AT</td>
<td>3074</td>
<td>3020474</td>
</tr>
<tr>
<td>HAS_MEMBER</td>
<td>14958</td>
<td>7530350</td>
</tr>
</tbody>
</table>

Table 6.3: Cardinalities of entities.

Value distribution The LDBC social network benchmark internally uses some different distributions to generate values. For example, the browsers, languages and gender used in the Person, Comment and Post entities are not uniformly distributed. This has an impact on the history generation process. The figures in Appendix B.1 show the distributions of the attributes used in the queries later on. They apply to dataset J from Table 6.2. Men and women are almost equally distributed as can be seen in Figure B.2. Figure B.3 shows the distribution of browsers in Comment and Post entities. Dates of birth are not very evenly distributed (see Figure B.4). This comes from the 2 stage generation process: The LDBC social network benchmark generator generates people with dates of birth in the eighties and nineties while the history generator evenly distributes dates of birth between 2014-08-25 and 2050-08-25. This looks a bit awkward, especially because people write messages before their actual dates of birth. It does not hurt the benchmark because dates of birth are not included at all in time related aspects in the queries and are just treated as yet another attribute. Figure B.5 shows the distribution of languages in Post entities.
6.2.6 Database Population

The database benchmark service either outputs the generated data into a directory as CSV files or populates a database directly via JDBC. We used the latter way to populate the system d and system m databases. For debugging and testing purposes, we used the dataset C from Table 6.2. We decided to use dataset J for the actual benchmark, because populating a system m database with that data set took several hours and populating a system d database took two days. This is due to the fact that no batch update mechanism can be used because the time stamps are generated from the database system upon transaction commit. First tests also showed it to be sufficient for running the queries in the experiments. To populate the Neo4j database from CSVs, we developed a small script in Neo4j’s scripting language.

6.3 Queries

To run the benchmark, we implemented the queries from section 5.1 for each system. They must be parameterizable and executable from the generic database benchmarking service.

6.3.1 Query Definition

All queries can be found in Appendix C. They were implemented for each database system and are sorted accordingly.

6.3.2 Parameters

The generic database benchmarking service allows parametrization of the queries. The parameters for the queries were either language independent, such as lengths of Comments or Posts or gender or had to be implemented individually for the query dialect of the database system. Table 6.4 shows the seven database independent parameters with their types and ranges. The values for the parameter BIRTHDAY_VARIABLE, GENDER, LANGUAGE, BROWSER_VARIABLE and LENGTH_VARIABLE are chosen uniformly at random from their respective domains. For PERSON_ID, PERSON_ID1, PERSON_ID2 and COMMENT_ID the benchmark framework has to lookup a random existing id in the database.

<table>
<thead>
<tr>
<th>parameter</th>
<th>type</th>
<th>domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIRTHDAY_VARIABLE</td>
<td>date</td>
<td>25.08.2014 to 25.08.2050</td>
</tr>
<tr>
<td>GENDER</td>
<td>string</td>
<td>male, female</td>
</tr>
<tr>
<td>LANGUAGE</td>
<td>string</td>
<td>en, zh, es, fr, ru, ar, pt, de, ur, ta, ml, te, mr, hi, as, bn, hu, it, sv, pl, af, tr, uk, ne, kk, fa, th, tk, ko, cs, qu, fi, uz, ny, si, be, ca, ru, to, ay, ku, mg, ha, eu, wr, wo, ln, co, gl, sr, az, pa, mk</td>
</tr>
<tr>
<td>BROWSER_VARIABLE</td>
<td>string</td>
<td>Safari, Internet Explorer, Chrome, Firefox, Opera</td>
</tr>
<tr>
<td>LENGTH_VARIABLE</td>
<td>integer</td>
<td>20 to 1500</td>
</tr>
<tr>
<td>PERSON_ID, PERSON_ID1, PERSON_ID2</td>
<td>long</td>
<td>chosen from existing person ids in the database</td>
</tr>
<tr>
<td>COMMENT_ID</td>
<td>long</td>
<td>chosen from existing comment ids in the database</td>
</tr>
</tbody>
</table>

Table 6.4: Database independent query parameters with domains.
**Time Parameters** The time stamps for *Point in Time* and *Time Range* queries had to be adopted to the particular database system: For Neo4j these are simple long values in a time range (see Listing C.2 and Listing C.3) whereas for system d and system m’s *Point in Time* queries it were UTC timestamps (see Listing C.5, Listing C.6 and Listing C.8). Since system m does not support *Time Range* queries natively, the time stamps for *Time Range* queries needed to be commit ids (see Listing C.9). Table 6.5 shows the ranges from which the parameters for the *Point in Time* and *Time Range* queries had to be chosen.

<table>
<thead>
<tr>
<th></th>
<th>validFrom</th>
<th>validTo</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Neo4j - simple longs</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Person</td>
<td>174001297</td>
<td>184001272</td>
</tr>
<tr>
<td>Comment</td>
<td>174001310</td>
<td>184001284</td>
</tr>
<tr>
<td>Post</td>
<td>1</td>
<td>184001038</td>
</tr>
<tr>
<td>knows</td>
<td>174001299</td>
<td>184001279</td>
</tr>
<tr>
<td>studyAt</td>
<td>174001294</td>
<td>184001275</td>
</tr>
<tr>
<td>hasMember</td>
<td>174001295</td>
<td>184001285</td>
</tr>
<tr>
<td><strong>System d - UTC time stamps</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Person</td>
<td>'2014-12-22 12:33:14'</td>
<td>'2014-12-23 12:10:41'</td>
</tr>
<tr>
<td>Comment</td>
<td>'2014-12-22 12:49:32'</td>
<td>'2014-12-23 12:09:11'</td>
</tr>
<tr>
<td>Post</td>
<td>'2014-12-22 13:12:00'</td>
<td>'2014-12-22 13:17:19'</td>
</tr>
<tr>
<td>knows</td>
<td>'2014-12-22 13:21:34'</td>
<td>'2014-12-23 12:10:49'</td>
</tr>
<tr>
<td>studyAt</td>
<td>'2014-12-22 13:36:55'</td>
<td>'2014-12-23 12:09:46'</td>
</tr>
<tr>
<td>hasMember</td>
<td>'2014-12-22 12:33:55'</td>
<td>'2014-12-23 11:56:10'</td>
</tr>
<tr>
<td><strong>System m - commit ids</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Person</td>
<td>32301586</td>
<td>50244771</td>
</tr>
<tr>
<td>Comment</td>
<td>313570</td>
<td>50244851</td>
</tr>
<tr>
<td>Post</td>
<td>314546</td>
<td>50244861</td>
</tr>
<tr>
<td>knows</td>
<td>314971</td>
<td>50244846</td>
</tr>
<tr>
<td>studyAt</td>
<td>315588</td>
<td>50244861</td>
</tr>
<tr>
<td>hasMember</td>
<td>312836</td>
<td>50244841</td>
</tr>
</tbody>
</table>

Table 6.5: Distributions of times after import of data set J into each particular system.
Only system d allows Time Range queries using the `FOR SYSTEM TIME BETWEEN ...AND ...
syntax. To simulate Time Range queries on Neo4j and system m, the WHERE clause had to be extended. Figure 6.5 shows data that is valid from `validFrom` to `validTo`. From the six queries using lower bound L and upper bound U, only queries one to four are allowed to return the tuple, because they overlap the data’s time span. The extended WHERE clauses then look like shown in Listing 6.5.

![Figure 6.5: Types of Time Range queries.](image)

**Listing 6.5: WHERE clause extension.**

```sql
MATCH ...
WHERE ...
AND ( (L <= data.validFrom AND data.validFrom <= U)
  OR (L <= data.validTo AND data.validTo <= U)
  OR (L >= data.validFrom AND data.validTo >= U)
)
...
```
6.3.3 Query Execution

To run the implemented queries the *generic database benchmarking service* needs drivers to communicate with the database system. System d and system m are already supported by the benchmarking service but Neo4j was not. This way we wrote a driver that extends the AbstractDriver class. The driver creates and maintains the JDBC connection to the Neo4j instance configured via the web interface. It also provides methods to make the lookups of the random parameters described in the section before. Figure 6.6 shows the implemented class.

![Class Diagram](image)

<table>
<thead>
<tr>
<th>AbstractDriver</th>
</tr>
</thead>
<tbody>
<tr>
<td>#connect(): java.sql.Connection</td>
</tr>
<tr>
<td>+commit(): void</td>
</tr>
<tr>
<td>+close(): void</td>
</tr>
<tr>
<td>+getIntegerFromQuery(String, String, ISchemaColumn): Integer</td>
</tr>
<tr>
<td>+getLongFromQuery(String, String, ISchemaColumn): Long</td>
</tr>
<tr>
<td>+getStringFromQuery(String, String, ISchemaColumn): String</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Neo4jDriver</th>
</tr>
</thead>
<tbody>
<tr>
<td>-connectionString: java.lang.String</td>
</tr>
<tr>
<td>#connect(): java.sql.Connection</td>
</tr>
<tr>
<td>+getIntegerFromQuery(String, String, ISchemaColumn): Integer</td>
</tr>
<tr>
<td>+getLongFromQuery(String, String, ISchemaColumn): Long</td>
</tr>
<tr>
<td>+getStringFromQuery(String, String, ISchemaColumn): String</td>
</tr>
</tbody>
</table>

Figure 6.6: Neo4j Driver.

6.4 Summary

In this chapter we show the details of the implementation of the benchmark. First we defined a data model, for which we generate stock and history data, and which is inserted into the *generic database benchmarking service*. Second we implemented the queries from chapter 5 and explained how they must be run using the benchmarking service. We also describe the details about the implemented classes for to make the benchmarking service run the queries using different parameters on the three database systems.
Chapter 7

Experiments

This chapter uses the previously defined graph benchmark to compare the three different database systems. First, we explain the experiments setup, including the machines used. After that, we show some optimization techniques on the used systems. At the end, we explain what exactly is measured, we do the measurements of the queries defined in chapter 5 and implemented in section 6.3, and analyze the results.

7.1 Setup

We ran the experiments on machines that have 24 GB memory, 4 Intel Xeon L5520 Processors with 4 cores each, and 1GBit Ethernet interconnect. The generic database benchmarking service was configured on a dedicated machine with the same specifications to keep the measurement overhead for queries minimal.

The experiments were run on three different storage engines: A Neo4j graph database, system d, system m.

Neo4j Graph Database  To run the experiments, we used the following Neo4j specific settings.

- Neo4j version 2.1.6 community edition
- Oracle JDK version 1.8.0_25
- 19GB Java heap space
- Neo4j index on following attributes:
  - Person.gender
  - Person.birthday
  - Comment.length
  - Post.length
  - Comment.browserUsed
  - Post.browserUsed
  - Post.language
  - ids on all entities
  - validFrom and validTo on all temporal entities
System d

- System d has native support for \textit{Point in Time} and \textit{Time Range} queries in system time and in application time. For the implemented queries we only use the system time dimension.

- History data is partitioned horizontally in system d. This means that current data is kept separated from past data in two different tables. The database unions both tables, if history queries are executed.

- We did not create any indices on system d.

System m

- System m has native support for \textit{Point in Time} queries in system time. We need to simulate \textit{Time Range} queries by constraining commit ids in the \texttt{WHERE} clauses as explained in section 6.3.2.

- History data is kept in a \textit{history table} in system m. This history table is partitioned into two parts: Currently visible data resides in the \textit{current}, while history data is saved in the \textit{history} partition.

- We did not create any indices on system m.
7.2 Optimization, Profiling and Query Plans

7.2.1 Neo4j Query Optimization

Early experiments showed some slow queries in Neo4j and after some research we found hints[6] that Neo4j’s query optimizer is not that mature to optimize the queries itself. Especially the queries that join several entities can be improved when using the referenced techniques. They say that Neo4j’s pattern matcher can handle shorter patterns better and one should break up long MATCH clauses into several subsequent ones. They also indicate that one should reduce the cardinalities of the sets passed from one MATCH clause to the next. Listing 7.1 shows a Cypher query which can be optimized to the one shown in Listing 7.2. The aforementioned technique is used in this case: The long MATCH clause is broken up into two shorter ones and the WITH DISTINCT p operation helps to reduce the cardinality of results used as input for the second part of the statement.

Listing 7.1: Standard query.

```
MATCH (o:Organisation)-[:STUDY_AT]-(p:Person)-[:STATE]-(:State {gender: "[GENDER]"})
RETURN DISTINCT o;
```

Listing 7.2: Optimized query.

```
MATCH (p:Person)-[:STATE]-(:State {gender: "[GENDER]"})
WITH DISTINCT p
MATCH (o:Organisation)-[:STUDY_AT]-(p)
RETURN distinct o;
```

Neo4j has the PROFILE keyword to generate the query plan and to count database accesses (DbHits) of a query. According to web sites [14] and [13], the number of DbHits must not be exceptionally high, since accessing the database and therefore accessing the disk was an expensive operation in Neo4j. Using the PROFILE keyword on the previous two queries reveals the output seen in Listing 7.3 and Listing 7.4. The total number of DbHits in the optimized version is about 25 times smaller compared to the standard version.

We used the aforementioned techniques upon the queries 1.2, that can be found in Appendix C. The queries are classified into NON, PIT and TR which stand for Non Temporal, Point in Time and Time Range. Details are not explained here because they have no influence on the optimization process and can be found in section 7.3.1. The execution times of these queries can be seen in Figure 7.1 and Figure 7.2. For these figures, the queries ran 10 times each. As those figures show, the optimizations do not always have positive effects. Not all queries can benefit from the optimizations: 1.2.b PIT is already fast and 1.2.b TR and 1.2.d NON have longer execution times. The queries 1.2.b NON, 1.2.d PIT and 1.2.d TR though can profit massively. Especially 1.2.d TR in the standard version ran into timeouts, which were set to 2000 seconds. All the different queries can be found and compared in Appendix C. For the experiments in the following sections always the faster version of each query was used.
Listing 7.3: Profiling of the standard query.

```
Distinct
  +Filter
    +TraversalMatcher

Operator | Rows | DbHits | Identifiers | Other
-----------------+-------+--------+-------------+---------------------+
  Distinct       | 7996  | 0      |             |                     |
  Filter         | 25357409 | 25357409 | hasLabel(o:Organisation(1)) |
  TraversalMatcher | 25357409 | 52261569 | o, UNNAMED22, o, UNNAMED44, o |
-----------------+-------+--------+-------------+---------------------+
```

Total database accesses: 77618978

Listing 7.4: Profiling of the optimized query.

```
Distinct(0)
  +Filter(0)
    +SimplePatternMatcher
      +Distinct(1)
        +Filter(1)
          +TraversalMatcher

Operator | Rows | DbHits | Identifiers | Other
-----------------+-------+--------+-------------+---------------------+
  Distinct(0)    | 7996  | 0      |             |                     |
  Filter(0)      | 747428 | 747428 | hasLabel(o:Organisation(1)) |
  SimplePatternMatcher | 747428 | 747428 | o, p, UNNAMED92 |
  Distinct(1)    | 12174 | 0      |             |                     |
  Filter(1)      | 309350 | 309350 | hasLabel(p:Person(4)) |
  TraversalMatcher | 309350 | 1237401 | p, UNNAMED16, p |
-----------------+-------+--------+-------------+---------------------+
```

Total database accesses: 3041607
Figure 7.1: Comparison of standard and optimized queries in Neo4j.

Figure 7.2: Comparison of standard and optimized queries in Neo4j.
7.2.2 Neo4j Index Usage

Neo4j has indices which were enabled for the attributes used in queries (see Listing 7.5). The indices though are not that good because, as explained in [8], they can only be used for equality constraints. To see whether Neo4j uses an index or not the PROFILE keyword can be used and its output can be analyzed. As an example, Listing 7.6 and Listing 7.7 show the profiles of two similar queries that differ only in the WHERE clause. The former uses the index and needs only 15 DbHits to find the final result of size 2 and thus returns in no time. The latter needs to scan all 24,059,989 State nodes to find its final 706 matching nodes which takes several seconds. So, determining whether an index is used can be done indirectly by looking at the amount of DbHits in the profiles.

Listing 7.5: Index creation

```sql
CREATE INDEX ON :State(birthday);
CREATE INDEX ON :State(gender);
CREATE INDEX ON :State(language);
CREATE INDEX ON :State(length);
CREATE INDEX ON :State(browserUsed);
CREATE INDEX ON :Organisation(type);
```

Listing 7.6: Index usage

```sql
PROFILE MATCH (c:Comment)-[:STATE]-(s:State)
WHERE s.length = 1000
RETURN c.id;
```

<table>
<thead>
<tr>
<th>Operator</th>
<th>Rows</th>
<th>DbHits</th>
<th>Identifiers</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>ColumnFilter</td>
<td>2</td>
<td>0</td>
<td>keep columns c.id</td>
<td></td>
</tr>
<tr>
<td>Extract</td>
<td>2</td>
<td>4</td>
<td>c.id</td>
<td></td>
</tr>
<tr>
<td>Filter</td>
<td>2</td>
<td>2</td>
<td>hasLabel(c:Comment(9))</td>
<td></td>
</tr>
<tr>
<td>TraversalMatcher</td>
<td>2</td>
<td>9</td>
<td>c, UNNAMED17, c</td>
<td></td>
</tr>
</tbody>
</table>

Total database accesses: 15
### Listing 7.7: No index usage

```plaintext
MATCH (c:Comment)-[:STATE]-(s:State )
WHERE s.length > 1000
RETURN c.id;
```

<table>
<thead>
<tr>
<th>Operator</th>
<th>Rows</th>
<th>DbHits</th>
<th>Identifiers</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>ColumnFilter</td>
<td>706</td>
<td>0</td>
<td></td>
<td>keep columns c.id</td>
</tr>
<tr>
<td>Extract</td>
<td>706</td>
<td>1412</td>
<td>c.id</td>
<td></td>
</tr>
<tr>
<td>Filter</td>
<td>706</td>
<td>2118</td>
<td>(hasLabel(s:State(10)) AND Property(s,length(7)) &gt; { AUTOINT0})</td>
<td>s, UNNAMED17, s</td>
</tr>
<tr>
<td>TraversalMatcher</td>
<td>706</td>
<td>24059989</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total database accesses: 24063519
CHAPTER 7. EXPERIMENTS

7.2.3 System d and System m Query Plans and Indexes

Query optimization  System d and system m both have query optimizers implemented. They use statistics and estimated costs to decide what the database operators and their call order would be, to execute queries most efficiently. Both systems allow getting optimizer plans which show the systems’ decision on those operators. System d also paints some character graphics showing the optimizer plan similar to Listing 7.8.

Listing 7.8: Statement with optimizer plan in system d

<table>
<thead>
<tr>
<th>Operator</th>
<th>(ID)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RETURN</td>
<td>( 1)</td>
</tr>
<tr>
<td>TBSCAN</td>
<td>( 2)</td>
</tr>
<tr>
<td>SORT</td>
<td>( 3)</td>
</tr>
<tr>
<td>UNION</td>
<td>( 4)</td>
</tr>
<tr>
<td>TBSCAN</td>
<td>( 5)</td>
</tr>
<tr>
<td>TBSCAN</td>
<td>( 6)</td>
</tr>
</tbody>
</table>

Table: Table:
| INST1 | INST1 |
| PERSON_HIST | PERSON |

These plans are often used to comprehend differences in query execution times. Especially when joining tables, one can see differences in the join strategies: Nested loop joins, merge sort joins and hash joins are used throughout the implemented queries. Nested loop joins loop for each tuple from the first relation over the whole second relation. Merge sort joins sort each relation and then loop over both relations once. Hash joins use hash tables to avoid rescanning whole relations which generally is fast. So when comparing query execution times on the same database system, it is often a good indicator to see which join technique was applied and thus be able to explain the speed differences.

Indices  Indices on the RDBMS were not applied at all. This is merely due to time constraints. To optimize slow queries, one should examine query plans and decide whether and which indices could help to improve execution times, similar to subsection 7.2.2. Since there were almost no queries which timed out on system d and system m, we focused on Neo4j, where the queries are not optimized in any way by the system and thus needed some manual care.
7.2.4 What Is Measured?

Execution time of a query is measured. Listing 7.9 shows the code which is executed for one measurement: First, a connection is made and a `Statement` object is created upon that. Second, we start the timer, execute the query and when the query returns, we read the whole result set. After reading the result, we stop the timer. This way we not only measure the query execution but also the data retrieval, which may have an impact on the measurements if the query result is very large. This is not generally bad because it represents a standard use case of JDBC in industry. We ran each query 25 times and measured its execution time. The figures in this chapter will show the mean execution times with their standard deviation. To let the systems warm up caches, we considered only the last 20 measurements for evaluation. According to the ranges defined in subsection 6.3.2, we set the query input parameters for each query execution newly at random.

Listing 7.9: Measurement code

```java
public void execute() throws SQLException{
    // get connection
    Connection connection = (Connection) driver.getConnection();

    // run query
    Statement statement = connection.createStatement();
    timer.start();
    int sum = 0;

    statement.executeQuery(queryText);
    ResultSet resultSet = statement.getResultSet();
    ResultSetMetaData md = resultSet.getMetaData();
    int countCol = md.getColumnCount();
    while (resultSet.next()){
        for(int i=1; i<=countCol; i++){
            String s = resultSet.getString(i);
            sum += (s != null) ? s.length() : 1;
        }
    }
    resultSet.close();
    timer.stop();
    statement.close();
    driver.close();
}
```
7.3 Measurements

7.3.1 Queries 1.1, 1.2, 1.3, 2.2, 2.3, and 3.3

As explained in subsection 5.5.1 the distinction between few temporal aspects and plenty temporal aspects for Time Range queries can be made. According to this distinction the queries 1.1, 1.2, 1.3, 2.2, 2.3 and 3.3 were measured and the results are shown in this section. The implementations of the queries can be found in Appendix C. Each of the following subsections contains three figures that show the execution times of the queries for Non Temporal (NON), Point in Time (PIT) and Time Range, few temporal aspects (TR).

Queries 1.1

The queries 1.1 only constrain on one specific attribute. 1.1.a does that on specific dates of birth while 1.1.d asks for people having a date of birth larger than a given date. Because of the gender attribute constrain, 1.1.b asks for about half of the tuples of Person which is a quite a large result set. 1.1.c constrains on one specific language only in the Post entity, which will return only a small result set. 1.1.e has to process an even larger entity, all Comments and Posts, where it asks for all messages larger than a random length. Figures 7.3, 7.4 and 7.5 show the execution times of the queries 1.1.

Findings

- Queries 1.1.a and 1.1.c (NON, PIT and TR) are fast on Neo4j. This is due to the fact, that these queries benefit from the indices on dates of birth and languages. 1.1.d and 1.1.e could not profit from the indices on dates of birth and lengths, because Neo4j indices are only beneficial for equality constraints.
- Query 1.1.b (NON, PIT and TR on Neo4j) takes time to return the large data set and to calculate distinct “Person” ids.
- On system d all NON queries except 1.1.e took about 5 seconds. 1.1.e took longer because of the large entities Comments and Posts that had to be processed. Query plans show that full table scans are made for all entities. I suspect that all queries could be improved significantly if indices were enabled for the required attributes.
- PIT constraints help system d and system m to improve the execution times compared to NON.
- TR constraints help system d to improve the execution times in compared to NON.
- TR constraints do not have much impact compared to NON on system m which is already fast.
- PIT or TR constraints do not help for Neo4j. We cannot observe much decrease in execution time when comparing NON, PIT and TR. This leaves room for improvement.
7.3. MEASUREMENTS

Figure 7.3: Measurements queries 1.1 Non Temporal (NON).

Figure 7.4: Measurements queries 1.1 Point in Time (PIT).

Figure 7.5: Measurements queries 1.1 Time Range (TR).
Queries 1.2

The queries 1.2 join two tables and constrain the result on one attribute. 1.2.a and 1.2.b join Universities with Persons where the first query constrains on a date of birth while the second only constrains on gender, which should result in a larger result set. 1.2.c joins only Posts with Persons and constrains on a language with an equality constraint. 1.2.d joins all Comments and Posts with Persons and constrains on lengths larger than a specified parameter. It returns a very large result set.

We optimized the queries 1.2.b and 1.2.d on Neo4j the way explained in subsection 7.2.1, otherwise some queries ran into timeouts.

Figures 7.6, 7.7 and 7.8 show the execution times of the queries 1.2.

Findings

- Comparing the query profiles of 1.2.a and 1.2.b on Neo4j shows many more database hits for 1.2.b than for 1.2.a. 1.2.b joins a much larger part of Person with University, because it only uses either females or males as constraint. 1.2.a uses a date of birth from a much broader range to constrain the Person entity and thus profits more from the index.

- Since system d has no index, it uses table scans, three nested loop joins and one hash join, which all make 1.2.a and 1.2.c slower than Neo4j. For 1.2.b system d decides to use two nested loop joins and two hash joins which in turn make it as fast as Neo4j.

- 1.2.c is generally fast in Neo4j because it is executed only on Posts written in the specific language, which can benefit from the index. It then only has to join this intermediate result with the Person entity.

- 1.2.d is the slowest on all systems, because it has to join the three largest entities, Person with Comment and Post, and constrain on length, where no index can help.

- PIT constraints do not help Neo4j much. 1.2.c PIT 1.2.d PIT even suffer compared to NON.

- TR constraints do not help Neo4j and system d compared to NON. 1.2.b TR on system d times out. Comparing the query plans of 1.2.b TR with NON reveal the already mentioned two hash joins in NON while TR tries to use four nested loop joins and thus fails.

- TR constraints do not influence system m much, since it is already fast.
Figure 7.6: Measurements queries 1.2 Non Temporal (NON).

Figure 7.7: Measurements queries 1.2 Point in Time (PIT).

Figure 7.8: Measurements queries 1.2 Time Range (TR).
CHAPTER 7. EXPERIMENTS

Queries 1.3
Queries 1.3 calculate aggregates over attributes of one specific entity constrained by an other attribute: 1.3.a simply counts all men or women in the database. 1.3.b counts messages larger than a given boundary and groups them by browsers. 1.3.c counts messages written in a given browser and groups them by length.

Figures 7.9, 7.10 and 7.11 show the execution times of the queries 1.3.

Findings

- Queries 1.3.a (NON, PIT and TR) are faster on all databases than the other queries. This is explainable by the fact that the first query does no grouping, and Person is the smaller entity than Comment unioned with Post.

- Queries 1.3.c on Neo4j (NON, PIT and TR) generally have a larger standard deviations than the other ones. This explainable because the browsers do not have the same distribution. This results in much faster queries for “Safari” than for “Firefox”.

- PIT constraints help system d a lot to improve the execution times compared to NON.

- PIT or TR constraints do not help for Neo4j. Not much decrease in execution time can be observed when comparing NON, PIT and TR. This leaves room for improvement.

- System m is generally the fastest.
Figure 7.9: Measurements queries 1.3 Non Temporal (NON).

Figure 7.10: Measurements queries 1.3 Point in Time (PIT).

Figure 7.11: Measurements queries 1.3 Time Range (TR).
CHAPTER 7. EXPERIMENTS

Queries 2.2
Queries 2.2 simply join two entities and return their result set. 2.2.a joins University with Person and 2.2.b joins Person with Comment and Post. Both will have large result sets while the second will be larger than the first. 2.2.a queries use DISTINCT to constrain the result set on all systems while the 2.2.b queries retrieve the whole data set.

Figures 7.12, 7.13 and 7.14 show the execution times of the queries 2.2.

Findings

- For the 2.2.a NON queries, using DISTINCT, system d and system m perform better than Neo4j.
- 2.2.a queries on system d can benefit more from PIT constraints than Neo4j or system m, which is already faster than the others.
- 2.2.a TR queries help system m and Neo4j compared to NON.
- 2.2.a TR on system d show a large standard deviation and a higher mean compared to NON. The result sets that differ hugely in cardinality are responsible for the large variance. They differ because of the differing sizes of the time ranges the query asks for. The higher mean compared to NON can be explained when looking at the query plans: NON uses two hash joins and two nested loop joins to join all tables while TR uses three loop joins and only one hash join.
- The 2.2.b NON and PIT queries on system d are worse than Neo4j. System m even runs out of memory when joining the two large tables, hence there are no measurements for that system.
- Neo4j and system d cannot profit from PIT constraints on 2.2.b queries. System m does not run into out-of-memory issues having these constraints, and is more than a magnitude faster than the other two database systems.
- TR constraints do not help Neo4j in 2.2.b compared to NON.
- TR constraints help 2.2.b on system d to perform better than NON. The query plans show that the database system decides to use a hash join for TR instead of a merge sort join as it did in NON and PIT.
- System m is still the fastest for the measurable queries, but is about one to two magnitudes slower than its corresponding 1.* queries.
- Interestingly, the DISTINCT queries show better performance than the ones without. One should inspect, whether the measurements with such huge result sets are really limited by the database performance, or if there are any other influences like the network or the JDBC driver.
7.3. MEASUREMENTS

Figure 7.12: Measurements queries 2.2 Non Temporal (NON).

Figure 7.13: Measurements queries 2.2 Point in Time (PIT).

Figure 7.14: Measurements queries 2.2 Time Range (TR).
Queries 2.3

Queries 2.3 aggregate a count of Person per University. Figures 7.15, 7.16 and 7.17 show the execution times of the queries 2.3.

Findings

- PIT and TR constraints do not improve Neo4j much.
- PIT constraints help system d but TR introduces a higher mean and a large variance in response time compared to NON. The variance comes from the differing result set sizes which differ because of the differing time slices. Comparing the query plans of the queries reveals three nested loop joins and one hash join for TR whereas PIT and NON have two nested loop joins and two hash joins. This explains the higher mean for TR.
- System m is always fast, with response times below 1 second.

Queries 3.3

Queries 3.3 calculate aggregates over one attribute of an entity: 3.3.a counts Comments and Posts grouped by browsers. 3.3.b counts people grouped by browsers and 3.3.c counts Comments and Posts grouped by length. Figures 7.18, 7.19 and 7.20 show the execution times of the queries 3.3.

Findings

- Queries 3.3.a and 3.3.c are always equally fast per system. This means that it does not matter whether the queries group into 5 different groups (browsers) or into about 500 different groups (lengths).
- Queries 3.3.b is always faster than their corresponding queries 3.3.a and 3.3.c. This comes from the different sizes of the entities the queries are working on: Comments unioned with Posts is larger than Person.
- PIT and TR constraints do not improve Neo4j.
- PIT and TR constraints help system d a lot.
- PIT and TR constraints help system m only slightly. System m is generally the fastest for these queries.
7.3. MEASUREMENTS

Figure 7.15: Measurements queries 2.3.a Non temporal (NON).

Figure 7.16: Measurements queries 2.3.a Point in Time (PIT).

Figure 7.17: Measurements queries 2.3.a Time Range (TR).
CHAPTER 7. EXPERIMENTS

Figure 7.18: Measurements queries 3.3 *Non Temporal (NON)*.

Figure 7.19: Measurements queries 3.3 *Point in Time (PIT)*.

Figure 7.20: Measurements queries 3.3 *Time Range (TR)*.
7.3.2 Queries with Graph Traversal

We measured the queries 1.4, 2.4, 3.4, 4.4, 1.5, 2.5, 5.5, 1.6, 2.6 and 6.6 for Non Temporal and Point in Time in Neo4j, and their results are shown in this section. There are queries that could not be implemented or measured due to the following limitations.

Limitations

No measurements on system d and system m  There are no experiments for system d and system m. Because these systems are standard RDBMS and use SQL as query language they do not bring a construct to express shortest paths between any types of tuples. Though these systems bring their own scripting engines to execute procedural code on the database there are no general purpose data structures like a priority queue available. Such a priority queue is necessary to implement a Dijkstra-Algorithm which would be sufficient to solve our shortest paths problems. It was decided not to invest more time on implementing a priority queue and shortest path algorithm out of the primitives the database scripting engines provide.

Neo4j’s Shortest Path  Neo4j has a function shortestPath, which takes a pattern as argument and returns the shortest path using the nodes and edges defined by that pattern. Listing 7.10 shows an example that finds the shortest path between two given people using an arbitrary number of KNOWS edges. This function is usually fast and returns in almost no time in our use cases. The downsides of this function are that only patterns of the form (node)-[EDGE*]-(node) are allowed and no additional constraints can be defined. For example, if we want to find the shortest path in the subgraph of men, we cannot pass the constraint “male” to that function. This means we have to take a detour as shown in Listing 7.11. This example lists all possible paths between person 1 and 2, orders them by length, checks if all nodes in the path contain men and return the first of these paths. We also constrain the length of the paths to be at most 5. If we did not constrain the length in a dense graph, queries using this technique would run into timeouts. This shows that this technique is way more expensive than using the built-in shortestPath function. The queries measured in this section use wherever possible the built-in function and fall back to the slow technique where it is not applicable.

Listing 7.10: Neo4j’s shortestPath function

| MATCH sp = shortestPath( (p1:Person {id:"1"})-[KNOWS *]-(p2:Person { id:"2" })) |
| RETURN sp |

Listing 7.11: Neo4j’s shortest path with slow technique

| MATCH path=(p1:Person { id:"1" })-[KNOWS +0..5]-(p2:Person { id:"2" }) |
| WITH path |
| ORDER BY length(path) |
| WHERE ALL (n in nodes(p) WHERE (n)-[:STATE]->(:State {gender:"male"})) |
| WITH path |
| LIMIT 1 |
| RETURN path as sp |
No measurements for *Time Range, plenty temporal aspects* To express *Temporal Shortest Paths*, which is the base for *Temporal Closeness Centrality* and *Temporal Network Diameter*, is not possible with Neo4j. A requirement of *Temporal Shortest Path* is that subsequent nodes or edges have at least overlapping time ranges, as explained in subsection 5.5.2. Neo4j’s Cypher query language has no possibility to examine two subsequent nodes or edges in a path, which would be necessary to implement *Temporal Shortest Path*. Listing 7.12 shows such a query that calculates the *Temporal Network Diameter*, where it would be necessary to implement *Temporal Shortest Path*: In the WHERE clause, always two subsequent relationships are compared whether they overlap, and only if they do, they are considered to be part of a temporal path. Because of that deficit, it was not possible to implement *Time Range, plenty temporal aspects* queries with Neo4j at all.

Listing 7.12: Query, which is not possible in Neo4j.

```cypher
MATCH p=(p1:Person)-[:KNOWS *0..4]-(p2:Person)
WHERE p1.id <> p2.id
AND ALL (
  ri, ri+1 in relationships(p):
  (ri.validFrom <= ri+1.validFrom AND ri+1.validFrom <= ri.validTo) 
  OR (ri.validFrom <= ri+1.validTo AND ri+1.validTo <= ri.validTo) 
  OR (ri.validFrom >= ri+1.validFrom AND ri+1.validFrom >= ri.validTo) 
)
WITH min(length(p)) as minlength, p1, p2
RETURN avg(minlength);
```
7.3. MEASUREMENTS

Queries 1.4, 2.4, 3.4 and 4.4

Query 1.4.a calculates shortest paths between two people in the subgraph of men. 2.4.a calculates shortest paths in the subgraph of Persons and Universities (2.4.a) or Forums respectively (2.4.b). 3.4.a calculates lengths of shortest paths between Persons knowing each other while 3.4.b sums lengths of messages along the shortest path from a Comment to its Post. 4.4 finally calculates a shortest path in the (Person)-(KNOWS)-(Person) graph. Purpose of these use cases are to examine execution times of traversal queries that have defined starting and ending nodes and should not need to process the whole graph.

Figures 7.21 and 7.22 show the execution times of the queries 1.4, 2.4, 3.4 and 4.4.

Findings

- Query 1.4.a is three to four magnitudes slower than the other queries for non temporal queries. This is because it was not possible to apply the shortestPath function on query 1.4.

- Queries 2.4.a, 2.4.b, 3.4.a and 4.4 NON use the shortestPath function and thus can return quickly.

- In the Point in Time queries it was not possible any more to use the shortestPath function because the time constraint cannot be passed to that function. Because of this, we used the slow technique and the queries unsurprisingly turned out much slower.

- Only query 3.4.b PIT remained fast compared to NON. This is because there is only one single path between Comment and its initial Post which is simple to detect for the database system. Note that the queries NON and PIT do not use the shortestPath function and have fast execution times.

- Comparing 2.4.a PIT with 2.4.b PIT shows that the latter is generally faster and has much more variance. This is because there are much fewer University than Forum nodes. So the subgraph for query 2.4.a is much denser than for 2.4.b, although it has about half the number of edges. More dense graphs mean many more paths to process using the slow technique, which in turn results in a higher mean execution time. In the less dense graph, it is possible that no path between two people can be found at all, which Neo4j can detect very quickly. This quick detection of non existing paths results in fast returning queries and introduces the large variance.
CHAPTER 7. EXPERIMENTS

Figure 7.21: Measurements queries 1.4, 2.4, 3.4 and 4.4 Non Temporal (NON).

Figure 7.22: Measurements queries 1.4, 2.4, 3.4 and 4.4 Point in Time (PIT).
7.3. MEASUREMENTS

Queries 1.5, 2.5 and 5.5

Queries 1.5 calculate the Closeness Centrality of a person in the Person/KNOWS subgraph. 1.5.a constrains gender to “male”, while 1.5.b constrains browsers. 2.5.a calculates the Closeness Centrality in the subgraph of people connected by STUDY_AT edges while 2.5.b does so using HAS_MEMBER edges. Finally 5.5 simply finds Closeness Centrality in the KNOWS subgraph using no constraints.

Purpose of these experiments was to examine execution times of queries that scan large parts of the graph and use shortest paths from one fixed node to all other nodes. To conduct these experiments, we used dataset C (6.9 MB) from Table 6.2. It was not possible to run any of the queries on a larger dataset because they time out. The size and densities of the graphs did not permit the queries to run in reasonable time. Because of the small size of the database, these measurements have to be taken with a grain of salt.

Figures 7.23 and 7.24 show the execution times of the queries 1.5, 2.5 and 5.5.

Findings

- These queries are generally slow and expensive to calculate. This is shown by the fact that we were forced to use the smallest possible dataset to run the queries on.
- Queries 2.5 NON and 5.5 NON profit from the shortestPath function while queries 1.5 NON had to use the slow technique to calculate shortest paths.
- For the Point in Time experiments, all queries had to be implemented using the slow technique to calculate shortest paths. This shows in the larger execution times compared to NON.
- Query 2.5.a PIT performed so well, because it had to process almost no subgraph. The subgraph of (Person) -[:STUDY_AT] - (University) -[:STUDY_AT] - (Person) is so small in the dataset C that for one point in time almost nothing was left and thus the query could return quickly.
Figure 7.23: Measurements queries 1.5, 2.5 and 5.5 Non Temporal (NON).

Figure 7.24: Measurements queries 1.5, 2.5 and 5.5 Point in Time (PIT).
Queries 1.6, 2.6 and 6.6

Query 1.6.a calculates the Network Diameter of all men in the KNOWS subgraph. 2.6.a and 2.6.b calculate the Network Diameter of the STUDY_AT and HAS_MEMBER subgraphs and 6.6 calculates it in the KNOWS subgraph without constraints. The purpose of these experiments is to examine the execution times of queries that have to scan the whole graph and possibly need to calculate Cartesian products between all nodes.

As in section 7.3.2, these experiments ran on the small dataset \( C \). Even with this small dataset, half of the queries timed out.

Figures 7.25 and 7.26 show the execution times of the queries 1.6, 2.6 and 6.6.

Findings

- These queries are generally slow and expensive to calculate. This shows through the fact that we were forced to use the smallest possible dataset to run the queries on.

- Queries 1.6 PIT and NON could not be implemented using Neo4j’s `shortestPath` function. Every measurement timed out at 2000 seconds.

- Queries 2.6.a, 2.6.b and 6.6 NON could profit from the `shortestPath` function.

- For the Point in Time experiments, all queries had to be implemented using the slow technique to calculate shortest paths. All but 2.6.a PIT timed out.

- When comparing the profiles of 2.6.a PIT with 2.6.b PIT it becomes clear that the former has much less DbHits than the latter. This comes from the fact that there are fewer universities than forums in the database and thus much fewer nodes need to be processed in 2.6.a PIT.
Figure 7.25: Measurements queries 1.6, 2.6 and 6.6 *Non Temporal (NON)*.

Figure 7.26: Measurements queries 1.6, 2.6 and 6.6 *Point in Time (PIT)*.
Chapter 8

Conclusion

In this thesis we identify use cases for temporal graphs and classify them into the three dimensions Queries Over Attributes, Graph Related Queries / Graph metrics, and Time Related Queries. Within these dimensions, we identify several query types that can be combined. Using these combinations, we introduce a benchmark that consists of a data generator and concrete queries within a social network example. In order to run these queries and to measure their execution times, we use the generic database benchmarking service. Therefore, we had to implement drivers, that communicate with the Neo4j graph database. During the implementation we learned how queries for Neo4j need to be optimized, and that we need to take care of the data structure, if we want to save a temporal graph in a graph database which does not yet support the temporal dimension natively.

Using the benchmark, we measured three different database systems. We summarize the results and findings in the following paragraphs.

Database Systems and Temporal Support

**System m** System m is generally faster than system d and Neo4j. This has a lot to do with it being an in-memory column store, not requiring to load data from disk. We cannot say much about its performance when temporal aspects are used, rather than “it is fast”. As future work, we have to increase the data volume by at least one order of magnitude and rerun the benchmark to be able to say more. For this, a machine with much more main memory needs to be used, since there is already a query (2.2.b NON) that cannot be executed due to out of memory exceptions on the current machines.

**System d** System d often gains speed from the Point in Time and Time Range constraints if it does not have to join tables. When joins come into play, like in the queries 1.2/2.2/2.3, things can get worse. Especially the Time Range queries suffer from using nested loop joins compared to hash joins in the Non Temporal versions.

**Neo4j** Neo4j is often on par or slightly faster than system d for Non Temporal queries but cannot benefit from Point in Time or Time Range constraints like system d does. It generally masters the join queries better than system d. This is not surprising since it only has to scan the existing edges and does not have to compute joins using the involved entities, which is done in the worst case through nested loop joins in system d. As future work, we have to examine why Neo4j does not benefit from the additional constraints. We also have to think about what would
be necessary to make things better, be it language extensions or generally make the database “time aware”. Maybe the ideas from the Timeline Index\[11\] can be adapted to graph databases.

Graph Traversal

**with RDBMS**  The benchmark implemented in this thesis suffers from not having traversal implementations for the RDBMSs. Future work will focus on such implementations to enable comparison with graph databases.

**with Neo4j**  One of Neo4j’s, or generally graph databases’, features is graph traversal. The experiments used graph traversal in the form of shortest path queries. The queries were either implemented using Neo4j’s shortestPath function or in a manual way. Using this shortestPath function is the preferred way since it is orders of magnitudes faster than the manual way. The shortestPath function lacks expressiveness since no constraints on a path query can be set or there is no way to compare two successive elements on path. These requirements are necessary to achieve better applicability, especially in our case with temporal queries.

Query Performance

**Indices with RDBMS**  One key feature of the relational database systems was left out in the experiments: Indices. In industry it is common to either rewrite slow queries or improve their speed with indices. This is often done when the system is running fully operational and becomes slower over time because certain queries become bottlenecks as the data volume increases. In this thesis, we started off by using no indices since the experiments ran at an acceptable speed. In future work, one will see whether system d can profit from indices, at least for the queries that constrain on attributes.

**Indices with Neo4j**  Neo4j only has rudimentary index support: The indices only support equality constraint comparisons on attributes. Because it was easy to apply, the indices were set in the database population script for the necessary attributes. The successful usage of the indices shows in the queries that were fast where they used equality comparisons. In future work, we will examine the internal index structure of Neo4j and find out how it could be improved to at least support inequality constraints. Temporal queries, the way they were implemented in this thesis, will probably benefit from such inequality constraint indices.

**Query Optimization**  RDBMS have the lead in automatic query optimization. The optimizers try to find the best operators according to the several factors. Neo4j has non of these things and needs manual care, which led to quite some trying until faster versions of queries could be found or to get to the insight that nothing can be done.

Summary  As the findings show, our benchmark contributes to finding and comprehending weak points of actual database systems, and helping graph database developers to improve their products. The main weak points we identified with our measurements are that Point in Time and Time Range queries using our simulated data model do not gain speed compared to Non Temporal queries in Neo4j. This can be improved in the future with better indices that have support for inequality constraints or by making the graph database fully “time aware”. This means that versions of nodes, edges, and attributes do not need a simulated temporal data model and will be versioned automatically, as today’s relational database systems already do.
Bibliography


Appendices
Appendix A

Listings for the Data Generator

Listing A.1: Helper function `deleteMessage`, recursively deletes a message and all its replies.

```plaintext
deleteMessage (m:Message):
    set m.validTo=CURRENT_TIME
    for each edge (m)<-[:REPLY_OF]-(c:Comment) where e.validTo=∞
        deleteMessage(c)
```

Listing A.2: Scenario Delete Message.

```plaintext
deleteMessage():
1. Uniform distribution of one of the following cases
   a) [probability=0.97]:
      uniformly at random select (m:Comment) where m.validTo=∞
   b) [probability=0.03]:
      uniformly at random select a (m:Post) where m.validTo=∞
2. deleteMessage(m)
```

Listing A.3: Scenario Update Comment.

```plaintext
updateComment():
1. uniformly at random select (m:Message) where m.validTo=∞
2. choose an attribute ATTR with the following distribution:
   a) [0.5] content (implies that the length changes as well)
   b) [0.3] browserUsed
   c) [0.2] locationIP
3. copy m with its attributes to new (m_new:Message)
4. set m.validTo=CURRENT_TIME // invalidates old message
5. set m_new.ATTR with new value,
    set m_new.validFrom=CURRENT_TIME
    set m_new.validTo=∞
```
Listing A.4: Scenario Delete Person.

deletePerson():
1. uniformly at random select (a:Person) where a.validTo=∞
2. set a.validTo=CURRENT_TIME
3. for each edge (a)-[e:]-(x) where e.validTo=∞:
   set e.validTo=CURRENT_TIME

Listing A.5: Scenario Update Person.

updatePerson():
1. uniformly at random select (a:Person) where a.validTo=∞
2. choose an attribute ATTR with the following distribution:
   a) [0.05] lastName
   b) [0.2] email
      with probability 0.5 create a new email
      with probability 0.5 change an existing email
   c) [0.2] speaks_language
      with probability 0.5 create a new speaks_language entry
      with probability 0.5 change an existing speaks_language entry
   d) [0.2] browserUsed
   e) [0.35] locationIP
3. copy a and its attributes to new (a_new:Person)
4. set a.validTo=CURRENT_TIME // invalidates old node
5. set a_new.ATTR with new value,
   set a_new.validFrom=CURRENT_TIME
   set a_new.validTo=∞

Listing A.6: Scenario Become Friends.

becomeFriends():
1. Uniform distribution of one of the following cases
   a) [probability=0.999]:
      uniformly at random select (a:Person) where a.validTo=∞
   b) [probability=0.001]:
      (a:Person)=createPerson()
2. Uniform distribution of one of the following cases
   a) [probability=0.999]:
      uniformly at random select (b:Person) where b.validTo=∞
      and b.id!=a.id and no edge (a)-[e:KNOWS]-(b) exists
   b) [probability=0.001]:
      (b:Person)=createPerson()
3. Add new edge (a)-[e:KNOWS]-(b),
   set e.validFrom=CURRENT_TIME,
   set e.validTo=∞
LISTINGS FOR THE DATA GENERATOR

Listing A.7: Scenario Quit Friendship.

```
quitFriendship():
  1. uniformly at random select a an edge
     (a:Person)-[e:KNOWS]-(b:Person) where e.validTo=∞
  2. update e.validTo=CURRENT_TIME
```

Listing A.8: Scenario Start Studying.

```
startStudying():
  1. Uniform distribution of one of the following cases
     a) [probability=0.999]:
        uniformly at random select (a:Person) where a.validTo=∞
     b) [probability=0.001]:
        (a:Person)=createPerson()
  2. uniformly at random select (u:University)
  3. create new edge (a)-[e:STUDY_AT]->(u)
     set e.validFrom=CURRENT_TIME,
     set e.validTo=∞
```

Listing A.9: Scenario Stop Studying.

```
stopStudying():
  1. uniformly at random select a an edge
     (u:University)<-[e:STUDY_AT]-(p:Person) where e.validTo=∞
  2. update e.validTo=CURRENT_TIME
```

Listing A.10: Scenario Become Forum Member.

```
becomeForumMember():
  1. Uniform distribution of one of the following cases
     a) [probability=0.999]:
        uniformly at random select (a:Person) where a.validTo=∞
     b) [probability=0.001]:
        (a:Person) = createPerson()
  2. uniformly at random select (f:Forum)
  3. create new edge (f)<-[e:HAS_MEMBER]-(a)
     set e.validFrom=CURRENT_TIME
     set e.validTo=∞
```
leaveForum():
1. uniformly at random select a an edge
   (f:Forum)<-[e:HAS_MEMBER]-(p:Person) where e.validTo=∞
2. update e.validTo=CURRENT_TIME
Appendix B

Figures for the Implementation Chapter
Figure B.1: Adapted UML diagram from LDBC Social Network Benchmark
B.1 Data Distribution of the Generated Datasets

Figure B.2: Distribution of gender in dataset J.
Figure B.3: Distribution of browsers in dataset J.

Figure B.4: A Histogram of dates of birth in dataset J.
Figure B.5: Distribution of languages in dataset J.
Appendix C

Benchmark Queries

Listing C.1: Queries *Non Temporal* with Neo4j Cypher.

```cypher
// 1.1
// a) MATCH (p:Person)-[:STATE]-(:State { birthday: "[BIRTHDAY_VARIABLE]"}) RETURN distinct p;
// b) MATCH (p:Person)-[:STATE]-(:State { gender: "[GENDER]" }) RETURN distinct p;
// c) MATCH (m)-[:STATE]-(:State {language: "[LANUGAGE]"}) WHERE (m:Comment OR m:Post) RETURN distinct m;
// d) MATCH (p:Person)-[:STATE]-(:State ) WHERE s.birthday > "[BIRTHDAY_VARIABLE]"
RETURN distinct p;
// e) MATCH (m)-[:STATE]-(:State) WHERE (m:Comment OR m:Post) AND s.length > [LENGTH_VARIABLE] RETURN distinct m;

// 1.2
// a) MATCH (o:Organisation)-[:STUDY_AT]-(p:Person)-[:STATE]-(:State { birthday: "[BIRTHDAY_VARIABLE]"})
RETURN DISTINCT o;
// b) standard MATCH (o:Organisation)-[:STUDY_AT]-(p:Person)-[:STATE]-(s:State) {gender: "[GENDER]"})
RETURN DISTINCT o;
// b) optimized MATCH (p:Person)-[:STATE]-(:State {gender: "[GENDER]"})
WITH distinct p
MATCH (o:Organisation)-[:STUDY_AT]-(p)
RETURN distinct o;
```
APPENDIX C. BENCHMARK QUERIES

// c)
MATCH (p:Person)-[:HAS_CREATOR]-(:Post)-[:STATE]-(:State {language:"[LANUGAGE]"})
RETURN DISTINCT p;

// d) standard
MATCH (p:Person)-[:HAS_CREATOR]-(:Post)-[:STATE]-(:State)
WHERE (m:Comment OR m:Post)
AND s.length > [LENGTH_VARIABLE]
RETURN DISTINCT p;

// d) optimized
MATCH (m)-[:STATE]-(:State)
WHERE (m:Comment OR m:Post)
AND s.length > [LENGTH_VARIABLE]
MATCH (p:Person)-[:HAS_CREATOR]-(:Post)
RETURN DISTINCT p;

// 1.3
// a)
MATCH (p:Person)-[:STATE]-(:State {gender:"[GENDER]"})
RETURN count(distinct p);

// b)
MATCH (m)-[:STATE]-(:State)
WHERE (m:Comment OR m:Post)
AND s.length > [LENGTH_VARIABLE]
RETURN s.browserUsed, count(m);

// c)
MATCH (m)-[:STATE]-(:State {browserUsed:"[BROWSER_VARIABLE]"})
WHERE (m:Comment OR m:Post)
RETURN s.length, count(m);

// 1.4
// a)
MATCH p=(pi:Person { id:"[PERSON_ID1]" })-[:KNOWS *0..5]-(:pj:Person { id:"[PERSON_ID2]" })
WITH p
ORDER BY length(p)
WHERE ALL (n in nodes(p) WHERE (n)-[:STATE]-(:State {gender:"male"})))
WITH p
LIMIT 1
UNWIND nodes(p) as n
MATCH (n)-[:STATE]-(:State)
RETURN distinct n, s.gender;

// 1.5
// a)
MATCH p=(pi:Person { id:"[PERSON_ID]" })-[:KNOWS *]-(:pj:Person)
WHERE ALL (n in nodes(p) WHERE (n)-[:STATE]-(:State {gender:"male"})))
WITH min(length(p)) as minlength, p
RETURN TOFLOAT(count(p)) / sum(minlength);

// b)
MATCH p=(pi:Person { id:"[PERSON_ID]" })-[:KNOWS *]-(:pj:Person)
WHERE ALL (n in nodes(p) WHERE (n)-[:STATE]-(:State {browserUsed:"[BROWSER_VARIABLE]"})))
WITH min(length(p)) as minlength, p
RETURN TOFLOAT(count(p)) / sum(minlength);
// 1.6
// a)
MATCH p=(p1:Person)-[:KNOWS *0..4]-(p2:Person)
WHERE p1.id <> p2.id AND ALL (n in nodes(p) WHERE (n)-[:STATE]-(:State {gender:"male"}))
WITH min(length(p)) as minlength, p1, p2
RETURN avg(minlength);

// 2.2
// a) changed
MATCH (p:Person)-[:STUDY_AT]->(u:Organisation { type:"university"})
RETURN p.id, u.id;

// b)
MATCH (m)-[:HAS_CREATOR]-(person:Person)
WHERE (m:Comment OR m:Post)
RETURN person.id, m.id;

// 2.3
// a)
MATCH (p:Person)-[:STUDY_AT]->(o:Organisation)
RETURN o.id, count(p);

// 2.4
// a)
MATCH p = shortestPath( (p1:Person {id:"[PERSON_ID1]"})-[:STUDY_AT*]-(p2:Person {id:"[PERSON_ID2]"}) )
WITH p
UNWIND nodes(p) as n
RETURN n, labels(n);

// b)
MATCH p=shortestPath((p1:Person {id:"[PERSON_ID1]"})-[:HAS_MEMBER*]-(p2:Person {id:"[PERSON_ID2]"}))
WITH p
UNWIND nodes(p) as n
RETURN n, labels(n);

// 2.5
// a)
MATCH p=shortestPath((pi:Person { id:"[PERSON_ID]"})-[:STUDY_AT*]-(pj:Person))
RETURN TOFLOAT(count(pj)) / sum(length(p));

// b)
MATCH p=shortestPath((pi:Person { id:"[PERSON_ID]"})-[:HAS_MEMBER*]-(pj:Person))
RETURN TOFLOAT(count(pj)) / sum(length(p));

// 2.6
// a)
MATCH p=shortestPath((p1:Person)-[:STUDY_AT *]-(p2:Person))
WHERE p1.id <> p2.id
RETURN avg(length(p));

// b)
MATCH p=shortestPath((p1:Person)-[:HAS_MEMBER *]-(p2:Person))
WHERE p1.id <> p2.id
RETURN avg(length(p));

// 3.3
// a)
APPENDIX C. BENCHMARK QUERIES

MATCH (m)-[s:STATE]-(state:State)
WHERE (m:Comment OR m:Post)
RETURN state.browserUsed, count(m);

// b)
MATCH (p:Person)-[s:STATE]-(state:State)
RETURN state.browserUsed, count(p);

// c)
MATCH (m)-[r:STATE]-(state:State)
WHERE (m:Comment OR m:Post)
RETURN state.length, count(distinct m);

//3.4
// a)
MATCH p = shortestPath((p1:Person { id:"[PERSON_ID1]" })-[:KNOWS *]-(p2:Person { id:"[PERSON_ID2]" }))
RETURN length(p);

// b)
MATCH sp = (c:Comment { id:"[COMMENT_ID]" })-[:REPLY_OF_COMMENT | REPLY_OF_POST *]->(p :Post)
WITH sp
UNWIND nodes(sp) as n
MATCH (n)-[s:STATE]-(s:State)
RETURN sum(s.length);

// 4.4
// a)
MATCH p = shortestPath((p1:Person { id: "[PERSON_ID1]" })-[:KNOWS *]-(p2:Person { id: "[PERSON_ID2]" }))
RETURN p

// 5.5
// a)
MATCH p=shortestPath((pi:Person { id:"[PERSON_ID]" })-[:KNOWS *]- (pj:Person))
RETURN TOFLOAT(count(pj)) / sum(length(p));

// 6.6 // very slow
// a)
MATCH p=shortestPath((p1:Person)-[:KNOWS *]-(p2:Person))
WHERE p1.id <> p2.id
RETURN avg(length(p));

// 1.1
// a)
MATCH (p:Person)-[r:STATE]-(:State { birthday: "[BIRTHDAY_VARIABLE]"})
WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo
RETURN DISTINCT p;

//b)
MATCH (p:Person)-[r:STATE]-(:State { gender: "[GENDER]" })
WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo
RETURN DISTINCT p;

//c)
MATCH (m)-[r:STATE]-(:State {language: "[LANUGAGE]"})

Listing C.2: Queries Point in Time with Neo4j Cypher.
WHERE (m:Comment OR m:Post)
    AND r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo
RETURN DISTINCT m;

// d)
MATCH (p:Person)-[r:STATE]-(s:State)
WHERE s.birthday > "[BIRTHDAY_VARIABLE]"
    AND r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo
RETURN distinct p;

// e)
MATCH (m)-[r:STATE]-(s:State)
WHERE (m:Comment OR m:Post)
    AND r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo
RETURN DISTINCT m;

// 1.2
// a)
MATCH (o:Organisation)-[r:STUDY_AT]-(p:Person)-[rr:STATE]-(:State { birthday: "[BIRTHDAY_VARIABLE]"})
WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo
    AND rr.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < rr.validTo
RETURN DISTINCT o;

// b) standard
MATCH (o:Organisation)-[r:STUDY_AT]-(p:Person)-[rr:STATE]-(:State {gender:"[GENDER]"})
WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo
    AND rr.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < rr.validTo
RETURN DISTINCT o;

// b) optimized
MATCH (p)-[rr:STATE]-(s:State {gender:'[GENDER]'})
WHERE rr.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < rr.validTo
MATCH (o:Organisation)-[r:STUDY_AT]-(p:Person)
WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo
RETURN distinct o.id;

// c)
MATCH (:State)-[rr:STATE]-(p:Person)-[:HAS_CREATOR]-(c:Post)-[r:STATE]-(:State {language:"[LANUGAGE]"})
WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo
    AND rr.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < rr.validTo
RETURN DISTINCT p;

// d) standard
MATCH (:State)-[rr:STATE]-(p:Person)-[:HAS_CREATOR]-(m)-[r:STATE]-(s:State)
WHERE (m:Comment OR m:Post)
    AND s.length > [LENGTH_VARIABLE]
    AND r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo
    AND rr.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < rr.validTo
RETURN DISTINCT p;

// d) optimized
MATCH (m)-[r:STATE]-(s:State)
WHERE (m:Comment OR m:Post)
    AND s.length > [LENGTH_VARIABLE]
    AND r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo
MATCH (p:Person)-[:HAS_CREATOR]-(m)
WITH distinct p
MATCH (p)-[rr:STATE]-(s:State)
WHERE rr.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < rr.validTo
RETURN distinct p;

// 1.3
// a)
MATCH (p:Person)-[r:STATE]-(:State {gender:"[GENDER]"})
WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo
RETURN count( distinct p);

// b)
MATCH (m)-[r:STATE]-(s:State)
WHERE (m:Comment OR m:Post)
   AND s.length > [LENGTH_VARIABLE]
   AND r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo
RETURN s.browserUsed, count(m);

// c)
MATCH (m)-[r:STATE]-(s:State {browserUsed:"[BROWSER_VARIABLE]"})
WHERE (m:Comment OR m:Post)
   AND r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo
RETURN s.length, count(m);

// 1.4
// a)
// MATCH p=shortestPath((p1:Person { id:"10995116283080" })-[:KNOWS*]-(p2:Person { id:"15393162801067" }})-[:KNOWS*0..4]-(p2:Person { id:"[PERSON_ID1]" })-[:KNOWS*0..4]-(pj:Person)
MATCH p=(pi:Person { id:"[PERSON_ID]" })-[:KNOWS*0..4]-(pj:Person)
WHERE ALL (n in nodes(p)
   WHERE (n)-[:STATE]-(:State {gender:"male"}))
AND ALL (r in relationships(p)
   WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo)
WITH p
ORDER BY length(p)
WHERE ALL (r in relationships(p)
   WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo)
   AND ALL (n in nodes(p)
      WHERE (n)-[:STATE]-(:State {gender:"male"}))
WITH p
LIMIT 1
UNWIND nodes(p) as n
MATCH (n)-[:STATE]-(s:State)
RETURN distinct n, s.gender;

// 1.5
// a)
MATCH p=(pi:Person { id:"[PERSON_ID]" })-[:KNOWS*0..4]-(pj:Person)
WHERE ALL (n in nodes(p)
   WHERE (n)-[:STATE]-(:State {gender:"male"}))
AND ALL (r in relationships(p)
   WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo)
WITH min(length(p)) as minlength, pj
RETURN TOFLOAT(count(pj)) / sum(minlength);

// b)
MATCH p=(pi:Person { id:"[PERSON_ID]" })-[:KNOWS*0..4]-(pj:Person)
WHERE ALL (n in nodes(p)
   WHERE (n)-[:STATE]-(:State {browserUsed:"[BROWSER_VARIABLE]"}))
AND ALL (r in relationships(p)
   WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo)
WITH min(length(p)) as minlength, pj
RETURN TOFLOAT(count(pj)) / sum(minlength);
// 1.6
// a)
MATCH p=(p1:Person)-[:KNOWS *0..4]-(p2:Person)
WHERE p1.id <> p2.id
AND ALL (n in nodes(p)
  WHERE (n)-[:STATE]->(:State {gender:"male"}))
AND ALL (r in relationships(p)
  WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo)
WITH min(length(p)) as minlength, p1, p2
RETURN avg(minlength);

// 2.2
// a)
MATCH (p:Person)-[r:STUDY_AT]->(u:Organisation { type:"university"})
WHERE r.validFrom < [POINT_IN_TIME_VAR]
  AND [POINT_IN_TIME_VAR] < r.validTo
RETURN p, u;

// b)
MATCH (:State) -[:STATE]-(:m)-[:HAS_CREATOR]-(person:Person)
WHERE (m:Comment OR m:Post)
  AND r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo
RETURN person, m;

// 2.3
// a)
MATCH (p:Person)-[r:STUDY_AT]-(o:Organisation)
WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo
return o, count(distinct p);

// 2.4
// a)
MATCH p=(p1:Person { id:"[PERSON_ID1]" })-[:STUDY_AT*0..6]-(p2:Person { id:"[PERSON_ID2]" })
WITH p
ORDER BY length(p)
WHERE ALL (r in relationships(p)
  WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo)
WITH p
LIMIT 1
UNWIND nodes(p) as n
RETURN n, labels(n);

// b)
MATCH p=(p1:Person { id:"[PERSON_ID1]" })-[:HAS_MEMBER*0..6]-(p2:Person { id:"[PERSON_ID2]" })
WITH p
ORDER BY length(p)
WHERE ALL (r in relationships(p)
  WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo)
WITH p
LIMIT 1
UNWIND nodes(p) as n
RETURN n, labels(n);

// 2.5
// a)
APPENDIX C. BENCHMARK QUERIES

MATCH p=(pi:Person { id:"[PERSON_ID]" })-[:STUDY_AT*0..4]-(pj:Person)
WHERE ALL (r in relationships(p)
    WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.
validTo)
WITH min(length(p)) as minlength, pj
RETURN TOFLOAT(count(pj)) / sum(minlength);

// b)
MATCH p=(pi:Person { id:"[PERSON_ID]" })-[:HAS_MEMBER*0..4]-(pj:Person)
WHERE ALL (r in relationships(p)
    WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.
validTo)
WITH min(length(p)) as minlength, pj
RETURN TOFLOAT(count(pj)) / sum(minlength);

// 2.6
// a)
MATCH p=(p1:Person)-[:STUDY_AT +0..4]-{p2:Person}
WHERE p1.id <> p2.id
    AND ALL (r in relationships(p)
        WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.
validTo)
WITH min(length(p)) as minlength, p1, p2
RETURN avg(minlength);

// b)
MATCH p=(p1:Person)-[:HAS_MEMBER +0..4]-{p2:Person}
WHERE p1.id <> p2.id
    AND ALL (r in relationships(p)
        WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo)
WITH min(length(p)) as minlength, p1, p2
RETURN avg(minlength);

// 3.3
// a)
MATCH {p}-[r:STATE]-(state:State)
WHERE (p:Comment OR p:Post)
    AND r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo
RETURN state.browserUsed, count(p);

// b)
MATCH {p:Person}-[r:STATE]-(state:State)
WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo
RETURN state.browserUsed, count(distinct p);

// c)
MATCH {m}-[r:STATE]-(state:State)
WHERE {m:Comment OR m:Post}
    AND r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo
RETURN state.length, count(distinct m);

// 3.4
// a)
MATCH p=(p1:Person { id:"[PERSON_ID1]" })-[[:KNOWS+0..4]-{p2:Person { id:"[PERSON_ID2]" }])
WITH p
ORDER BY length(p)
WHERE ALL (r in relationships(p) WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo)
WITH p
LISTING C.3: QUERIES TIME RANGE, FEW TEMPORAL ASPECTS WITH NEO4J CYPER.

// 1.1
// a)
MATCH (p:Person)-[r:STATE]-(:State (birthday: "[BIRTHDAY_VARIABLE]"))
WHERE ([POINT_IN_TIME_L_BOUND] <= r.validFrom AND r.validFrom <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
    OR ([POINT_IN_TIME_L_BOUND] <= r.validTo AND r.validTo <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
    OR ([POINT_IN_TIME_L_BOUND] >= r.validFrom AND r.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
RETURN DISTINCT p;

// 1.1
// b)
// there is no shortest path, there is only one path
MATCH sp = (c:Comment { id: "[COMMENT_ID]" })-[[:REPLY_OF_COMMENT | REPLY_OF_POST *]]->(p:Post)
WITH sp
ORDER BY length(sp)
UNWIND nodes(sp) as n
MATCH (n)-[r:STATE]-(s:State) WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo
RETURN sum(s.length);

// 4.4
// a)
MATCH p = (p1:Person { id: "[PERSON_ID1]" })-[[:KNOWS *0..7]]-(p2:Person { id: "[PERSON_ID2]" })
WITH p
ORDER BY length(p)
WHERE ALL (r in relationships(p) WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo)
WITH p
LIMIT 1
UNWIND nodes(p) as n
RETURN distinct n;

// 5.5
// a)
MATCH p = (p1:Person { id: "[PERSON_ID]" })-[[:KNOWS *0..4]]-(p2:Person)
WHERE ALL (r in relationships(p) WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo)
WITH min(length(p)) as minlength, p1, p2
RETURN TOFLOAT(count(p2)) / sum(minlength);

// 6.6 // very slow
// a)
MATCH p = (p1:Person)-[:KNOWS *0..4]-(p2:Person)
WHERE p1.id <> p2.id
    AND ALL (r in relationships(p) WHERE r.validFrom < [POINT_IN_TIME_VAR] AND [POINT_IN_TIME_VAR] < r.validTo)
WITH min(length(p)) as minlength, p1, p2
RETURN avg(minlength);
APPENDIX C. BENCHMARK QUERIES

// b)
MATCH (p:Person)-[r:STATE]-(s:State { gender: "[GENDER]" })
WHERE (
  ([POINT_IN_TIME_L_BOUND] <= r.validFrom AND r.validFrom <= 
    [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
  OR ([POINT_IN_TIME_L_BOUND] <= r.validTo AND r.validTo <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
  OR ([POINT_IN_TIME_L_BOUND] >= r.validFrom AND r.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
)
RETURN DISTINCT p;

// c)
MATCH (m)-[r:STATE]-(s:State { language: "[LANUGAGE]"})
WHERE (m:Comment OR m:Post)
AND (
  ([POINT_IN_TIME_L_BOUND] <= r.validFrom AND r.validFrom <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
  OR ([POINT_IN_TIME_L_BOUND] <= r.validTo AND r.validTo <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
  OR ([POINT_IN_TIME_L_BOUND] >= r.validFrom AND r.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
)
RETURN DISTINCT m;

// d)
MATCH (p:Person)-[r:STATE]-(s:State)
WHERE s.birthday > "[BIRTHDAY_VARIABLE]"
AND (
  ([POINT_IN_TIME_L_BOUND] <= r.validFrom AND r.validFrom <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
  OR ([POINT_IN_TIME_L_BOUND] <= r.validTo AND r.validTo <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
  OR ([POINT_IN_TIME_L_BOUND] >= r.validFrom AND r.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
)
RETURN DISTINCT p;

// e)
MATCH (m)-[r:STATE]-(s:State)
WHERE (m:Comment OR m:Post)
AND s.length > [LENGTH_VARIABLE]
AND (
  ([POINT_IN_TIME_L_BOUND] <= r.validFrom AND r.validFrom <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
  OR ([POINT_IN_TIME_L_BOUND] <= r.validTo AND r.validTo <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
  OR ([POINT_IN_TIME_L_BOUND] >= r.validFrom AND r.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
)
RETURN DISTINCT m;

// 1.2
// a)
MATCH (o:Organisation)-[r:STUDY_AT]-(p:Person)-[rr:STATE]-(s:State { birthday: "[BIRTHDAY_VARIABLE]" })
WHERE (
  ([POINT_IN_TIME_L_BOUND] <= r.validFrom AND r.validFrom <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
  OR ([POINT_IN_TIME_L_BOUND] <= r.validTo AND r.validTo <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
  OR ([POINT_IN_TIME_L_BOUND] >= r.validFrom AND r.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
)
AND (  ([POINT_IN_TIME_L_BOUND] <= rr.validFrom AND rr.validFrom <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
  OR ([POINT_IN_TIME_L_BOUND] <= rr.validTo AND rr.validTo <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
  OR ([POINT_IN_TIME_L_BOUND] >= rr.validFrom AND rr.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
)
OR ([POINT_IN_TIME_L_BOUND] >= rr.validFrom AND rr.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
)
RETURN DISTINCT o;

// b) standard
MATCH (o:Organisation)-[r:STUDY_AT]-(:Person)-[rr:STATE]-(:State {gender:'[GENDER]'})
WHERE ( ([POINT_IN_TIME_L_BOUND] <= r.validFrom AND r.validFrom <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
    OR ([POINT_IN_TIME_L_BOUND] <= r.validTo AND r.validTo <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
    OR ([POINT_IN_TIME_L_BOUND] >= r.validFrom AND r.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
)
AND ( ( [POINT_IN_TIME_L_BOUND] <= rr.validFrom AND rr.validFrom <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
    OR ([POINT_IN_TIME_L_BOUND] <= rr.validTo AND rr.validTo <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
    OR ([POINT_IN_TIME_L_BOUND] >= rr.validFrom AND rr.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
)
RETURN DISTINCT o;

// b) optimized
MATCH (p:Person)-[rr:STATE]-(:State {gender:'[GENDER]'})
WHERE ( ([POINT_IN_TIME_L_BOUND] <= rr.validFrom AND rr.validFrom <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
    OR ([POINT_IN_TIME_L_BOUND] <= rr.validTo AND rr.validTo <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
    OR ([POINT_IN_TIME_L_BOUND] >= rr.validFrom AND rr.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
)
MATCH (o:Organisation)-[r:STUDY_AT]-(p)
WHERE ( ([POINT_IN_TIME_L_BOUND] <= r.validFrom AND r.validFrom <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
    OR ([POINT_IN_TIME_L_BOUND] <= r.validTo AND r.validTo <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
    OR ([POINT_IN_TIME_L_BOUND] >= r.validFrom AND r.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
)
RETURN DISTINCT o;

// c)
MATCH (:State)-[rr:STATE]-(p:Person)-[:HAS_CREATOR]-(c:Post)-[r:STATE]-(:State {language:'[LANGUAGE]'})
WHERE ( ([POINT_IN_TIME_L_BOUND] <= r.validFrom AND r.validFrom <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
    OR ([POINT_IN_TIME_L_BOUND] <= r.validTo AND r.validTo <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
    OR ([POINT_IN_TIME_L_BOUND] >= r.validFrom AND r.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
)
AND ( ( [POINT_IN_TIME_L_BOUND] <= rr.validFrom AND rr.validFrom <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
    OR ([POINT_IN_TIME_L_BOUND] <= rr.validTo AND rr.validTo <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
    OR ([POINT_IN_TIME_L_BOUND] >= rr.validFrom AND rr.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
)
RETURN DISTINCT p;

// d) standard
MATCH (:State)-[rr:STATE]-(p:Person)-[:HAS_CREATOR]-(m)-[r:STATE]-(s:State)
WHERE (m:Comment OR m:Post)
APPENDIX C. BENCHMARK QUERIES

AND s.length > [LENGTH_VARIABLE]
AND ( ([POINT_IN_TIME_L_BOUND] <= r.validFrom AND r.validFrom <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
OR ([POINT_IN_TIME_L_BOUND] <= r.validTo AND r.validTo <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
OR ([POINT_IN_TIME_L_BOUND] >= r.validFrom AND r.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
)
AND ( ([POINT_IN_TIME_L_BOUND] <= rr.validFrom AND rr.validFrom <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
OR ([POINT_IN_TIME_L_BOUND] <= rr.validTo AND rr.validTo <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
OR ([POINT_IN_TIME_L_BOUND] >= rr.validFrom AND rr.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
)
RETURN DISTINCT p;

// d) optimized
MATCH (m)-[r:STATE]-(s:State)
WHERE (m:Comment OR m:Post)
AND s.length > [LENGTH_VARIABLE]
AND ( ([POINT_IN_TIME_L_BOUND] <= r.validFrom AND r.validFrom <= [POINT_IN_TIME_LBOUND] + [DIFFERENCE])
OR ([POINT_IN_TIME_L_BOUND] <= r.validTo AND r.validTo <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
OR ([POINT_IN_TIME_L_BOUND] >= r.validFrom AND r.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
)
RETURN DISTINCT p

MATCH (m)-[r:STATE]-(s:State)
WHERE (m:Comment OR m:Post)
AND s.length > [LENGTH_VARIABLE]
AND ( ([POINT_IN_TIME_L_BOUND] <= r.validFrom AND r.validFrom <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
OR ([POINT_IN_TIME_L_BOUND] <= r.validTo AND r.validTo <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
OR ([POINT_IN_TIME_L_BOUND] >= r.validFrom AND r.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
)
RETURN DISTINCT p

// 1.3
// a)
MATCH (p:Person)-[r:STATE]-(s:State {gender:"[GENDER]"})
WHERE ([POINT_IN_TIME_L_BOUND] <= r.validFrom AND r.validFrom <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
OR ([POINT_IN_TIME_L_BOUND] <= r.validTo AND r.validTo <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
OR ([POINT_IN_TIME_L_BOUND] >= r.validFrom AND r.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
RETURN count( DISTINCT p);

// b)
MATCH (m)-[r:STATE]-(s:State)
WHERE (m:Comment OR m:Post)
AND s.length > [LENGTH_VARIABLE]
AND ( ([POINT_IN_TIME_L_BOUND] <= r.validFrom AND r.validFrom <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
OR ([POINT_IN_TIME_L_BOUND] <= r.validTo AND r.validTo <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
OR ([POINT_IN_TIME_L_BOUND] >= r.validFrom AND r.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
)
RETURN s.browserUsed, count(m);

// c)
MATCH (m:Comment OR m:Post)
WHERE (m:Comment OR m:Post)
AND (([POINT_IN_TIME_L_BOUND] <= r.validFrom AND r.validFrom <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
 OR ([POINT_IN_TIME_L_BOUND] <= r.validTo AND r.validTo <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
 OR ([POINT_IN_TIME_L_BOUND] >= r.validFrom AND r.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE]))
RETURN s.length, count(m);

// 2.2
// a)
MATCH (p:Person)-[r:STUDY_AT]->(u:Organisation { type:"university"})
WHERE (p:Person)-[r:STUDY_AT]->(u:Organisation { type:"university"})
WHERE ([POINT_IN_TIME_L_BOUND] <= r.validFrom AND r.validFrom <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
 OR ([POINT_IN_TIME_L_BOUND] <= r.validTo AND r.validTo <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
 OR ([POINT_IN_TIME_L_BOUND] >= r.validFrom AND r.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
RETURN p, u;

// b)
MATCH (:State)-[r:STATE]-(m:Comment)-[:HAS_CREATOR]-(person:Person)
WHERE (:State)-[r:STATE]-(m:Comment)-[:HAS_CREATOR]-(person:Person)
WHERE ([POINT_IN_TIME_L_BOUND] <= r.validFrom AND r.validFrom <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
 OR ([POINT_IN_TIME_L_BOUND] <= r.validTo AND r.validTo <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
 OR ([POINT_IN_TIME_L_BOUND] >= r.validFrom AND r.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
RETURN person, m;

//2.3
// a)
MATCH (p:Person)-[r:STUDY_AT]->(o:Organisation)
WHERE (p:Person)-[r:STUDY_AT]->(o:Organisation)
WHERE ([POINT_IN_TIME_L_BOUND] <= r.validFrom AND r.validFrom <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
 OR ([POINT_IN_TIME_L_BOUND] <= r.validTo AND r.validTo <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
 OR ([POINT_IN_TIME_L_BOUND] >= r.validFrom AND r.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
RETURN o, count(distinct p);

// 3.3
// a)
MATCH (p:Comment OR p:Post)
WHERE (p:Comment OR p:Post)
AND (([POINT_IN_TIME_L_BOUND] <= r.validFrom AND r.validFrom <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
 OR ([POINT_IN_TIME_L_BOUND] <= r.validTo AND r.validTo <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
 OR ([POINT_IN_TIME_L_BOUND] >= r.validFrom AND r.validTo >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE]))
RETURN state.browserUsed, count(p);

// b)
MATCH (p:Person)-[r:STATE]-(state:State)
WHERE (p:Person)-[r:STATE]-(state:State)
WHERE ([POINT_IN_TIME_L_BOUND] <= r.validFrom AND r.validFrom <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
APPENDIX C. BENCHMARK QUERIES

```
OR ([POINT_IN_TIME_L_BOUND] <= r.validTo AND r.validTo <= [
    POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
OR ([POINT_IN_TIME_L_BOUND] >= r.validFrom AND r.validTo >= [
    POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
RETURN state.browserUsed, count(p);
```

Listing C.4: Queries Non Temporal with System d SQL.

```
-- 1.1
-- a)
SELECT distinct p_union.id
FROM person FOR SYSTEM_TIME
FROM '0001-01-01' TO '9999-12-30'
AS p_union
WHERE p_union.birthday = '[BIRTHDAY_VARIABLE]';

-- b)
SELECT distinct p_union.id
FROM person FOR SYSTEM_TIME
FROM '0001-01-01' TO '9999-12-30'
AS p_union
WHERE p_union.gender = '[GENDER]';

-- c)
SELECT distinct p_union.id
FROM post FOR SYSTEM_TIME
FROM '0001-01-01' TO '9999-12-30'
AS p_union
WHERE p_union.language = '[Lanugage]';

-- d)
SELECT distinct p_union.id
FROM person FOR_SYSTEM_TIME
FROM '0001-01-01' TO '9999-12-30'
AS p_union
WHERE p_union.birthday > '[BIRTHDAY_VARIABLE]';

-- e)
SELECT distinct m_union.id
FROM (SELECT id,length FROM comment FOR SYSTEM_TIME
FROM '0001-01-01' TO '9999-12-30'
UNION
SELECT id,length FROM post FOR SYSTEM_TIME
FROM '0001-01-01' TO '9999-12-30')
AS m_union
WHERE m_union.length > [LENGTH_VARIABLE];
```

```
-- 1.2
-- a)
SELECT distinct o_union.organisation_id
FROM person FOR SYSTEM_TIME
FROM '0001-01-01' TO '9999-12-30'
AS o_union
WHERE o_union.person_id = p_union.id
AND p_union.birthday = '[BIRTHDAY_VARIABLE]';

-- b)
SELECT distinct o_union.organisation_id
FROM person FOR SYSTEM_TIME
FROM '0001-01-01' TO '9999-12-30'
AS p_union,
```
-- c) SELECT distinct p_union.id
FROM post FOR SYSTEM_TIME FROM '0001-01-01' TO '9999-12-30' AS m_union,
person FOR SYSTEM_TIME FROM '0001-01-01' TO '9999-12-30' AS p_union
WHERE m_union.creator_id = p_union.id
AND m_union.language = '[LANGUAGE]';

-- d) SELECT distinct p_union.id
FROM (SELECT id,creator_id
FROM comment FOR SYSTEM_TIME FROM '0001-01-01' TO '9999-12-30'
WHERE length > [LENGTH_VARIABLE]
UNION
SELECT id,creator_id
FROM post FOR SYSTEM_TIME FROM '0001-01-01' TO '9999-12-30'
WHERE length > [LENGTH_VARIABLE])
AS m_union,
person FOR SYSTEM_TIME FROM '0001-01-01' TO '9999-12-30' AS p_union
WHERE m_union.creator_id = p_union.id;

-- 1.3
-- a) SELECT count( distinct p_union.id)
FROM person FOR SYSTEM_TIME FROM '0001-01-01' TO '9999-12-30' AS p_union
WHERE p_union.gender = '[GENDER]';

-- b) SELECT m_union.browserused, count(m_union.id)
FROM (SELECT id,length,browserused FROM comment FOR SYSTEM_TIME FROM '0001-01-01'
TO '9999-12-30'
UNION
SELECT id,length,browserused FROM post FOR SYSTEM_TIME FROM '0001-01-01' TO '9999-12-30')
AS m_union
WHERE m_union.length > [LENGTH_VARIABLE]
GROUP BY m_union.browserused;

-- c) SELECT m_union.length, count(m_union.id) cnt
FROM (SELECT id,length,browserused FROM comment FOR SYSTEM_TIME FROM '0001-01-01'
TO '9999-12-30'
UNION
SELECT id,length,browserused FROM post FOR SYSTEM_TIME FROM '0001-01-01' TO '9999-12-30')
AS m_union
WHERE m_union.browserused = '[BROWSER_VARIABLE]'
GROUP BY m_union.length;

-- 2.2
-- a) SELECT distinct p_union.id, o_union.organisation_id
FROM person FOR SYSTEM_TIME FROM '0001-01-01' TO '9999-12-30' AS p_union,
person_studyat_organisation FOR SYSTEM_TIME FROM '0001-01-01' TO '9999-12-30' AS o_union
WHERE o_union.person_id = p_union.id

-- b)
SELECT distinct m_union.id, p_union.id
FROM (SELECT id,creator_id FROM comment FOR SYSTEM_TIME FROM '0001-01-01' TO '9999-12-30'
UNION
SELECT id,creator_id FROM post FOR SYSTEM_TIME FROM '0001-01-01' TO '9999-12-30') AS m_union,
person FOR SYSTEM_TIME FROM '0001-01-01' TO '9999-12-30' AS p_union
WHERE m_union.creator_id = p_union.id;

-- 2.3
-- a)
SELECT o_union.organisation_id, count(p_union.id) cnt
FROM person FOR SYSTEM_TIME FROM '0001-01-01' TO '9999-12-30' AS p_union,
person_studyat_organisation FOR SYSTEM_TIME FROM '0001-01-01' TO '9999-12-30' AS o_union
WHERE o_union.person_id = p_union.id
GROUP BY o_union.organisation_id;

-- 3.3
-- a)
SELECT m_union.browserused, count(m_union.id)
FROM (SELECT id,browserused FROM comment FOR SYSTEM_TIME FROM '0001-01-01' TO '9999-12-30'
UNION
SELECT id,browserused FROM post FOR SYSTEM_TIME FROM '0001-01-01' TO '9999-12-30') AS m_union
GROUP BY m_union.browserused;

-- b)
SELECT p_union.browserused, count(p_union.id)
FROM person FOR SYSTEM_TIME FROM '0001-01-01' TO '9999-12-30' AS p_union
GROUP BY p_union.browserused;

-- c)
SELECT m_union.length, count(m_union.id)
FROM (SELECT id,length FROM comment FOR SYSTEM_TIME FROM '0001-01-01' TO '9999-12-30'
UNION
SELECT id,length FROM post FOR SYSTEM_TIME FROM '0001-01-01' TO '9999-12-30') AS m_union
GROUP BY m_union.length;

Listing C.5: Queries Point in Time with System d SQL.

-- 1.1
-- a)
SELECT distinct p_union.id
FROM person FOR SYSTEM_TIME AS OF '[POINT_IN_TIME_VAR]' AS p_union
WHERE p_union.birthday = '[BIRTHDAY_VARIABLE]';

-- b)
SELECT distinct p_union.id
FROM person FOR SYSTEM_TIME AS OF '[POINT_IN_TIME_VAR]' AS p_union
WHERE p_union.gender = '[GENDER]';

-- c)
SELECT distinct p_union.id
FROM post FOR SYSTEM_TIME AS OF '[POINT_IN_TIME_VAR]' AS p_union
WHERE p_union.language = '[LANUGAGE]';
-- d)
SELECT distinct p_union.id
FROM person FOR SYSTEM_TIME AS OF 'POINT_IN_TIME_VAR' AS p_union
WHERE p_union.birthday > BIRTHDAY_VARIABLE;

-- e)
SELECT distinct m_union.id
FROM (SELECT id,length FROM comment FOR SYSTEM_TIME AS OF 'POINT_IN_TIME_VAR'
UNION
SELECT id,length FROM post FOR SYSTEM_TIME AS OF 'POINT_IN_TIME_VAR')
AS m_union
WHERE m_union.length > LENGTH_VARIABLE;

-- 1.2
-- a)
SELECT distinct o_union.organisation_id
FROM person FOR SYSTEM_TIME AS OF 'POINT_IN_TIME_VAR' AS p_union,
person_studyat_organisation FOR SYSTEM_TIME AS OF 'POINT_IN_TIME_VAR' AS o_union
WHERE o_union.person.id = p_union.id
AND p_union.birthday = BIRTHDAY_VARIABLE;

-- b)
SELECT distinct o_union.organisation_id
FROM person FOR SYSTEM_TIME AS OF 'POINT_IN_TIME_VAR' AS p_union,
person_studyat_organisation FOR SYSTEM_TIME AS OF 'POINT_IN_TIME_VAR' AS o_union
WHERE o_union.person.id = p_union.id
AND p_union.gender = GENDER;

-- c)
SELECT distinct p_union.id
FROM post FOR SYSTEM_TIME AS OF 'POINT_IN_TIME_VAR' AS m_union,
person FOR SYSTEM_TIME AS OF 'POINT_IN_TIME_VAR' AS p_union
WHERE m_union.creator_id = p_union.id
AND m_union.language = LANGUAGE;

-- d)
SELECT distinct p_union.id
FROM (SELECT id,creator_id
     FROM comment FOR SYSTEM_TIME AS OF 'POINT_IN_TIME_VAR'
     WHERE length > LENGTH_VARIABLE
UNION
SELECT id,creator_id FROM post FOR SYSTEM_TIME AS OF 'POINT_IN_TIME_VAR'
     WHERE length > LENGTH_VARIABLE)
AS m_union,
person FOR SYSTEM_TIME AS OF 'POINT_IN_TIME_VAR' AS p_union
WHERE m_union.creator_id = p_union.id;

-- 1.3
-- a)
SELECT count(distinct p_union.id)
FROM person FOR SYSTEM_TIME AS OF 'POINT_IN_TIME_VAR' AS p_union
WHERE p_union.gender = GENDER;

-- b)
SELECT m_union.browserused, count(m_union.id)
FROM (SELECT id,length,browserused FROM comment FOR SYSTEM_TIME AS OF 'POINT_IN_TIME_VAR'
     UNION

APPENDIX C. BENCHMARK QUERIES

```sql
-- c) SELECT m_union.length, count(m_union.id) cnt FROM (SELECT id, length, browserused FROM comment FOR SYSTEM_TIME AS OF '[POINT_IN_TIME_VAR]') AS m_union WHERE m_union.browserused = '[BROWSER_VARIABLE]' GROUP BY m_union.length;

-- 2.2
-- a)
SELECT distinct p_union.id, o_union.organisation_id FROM person FOR SYSTEM_TIME AS OF '[POINT_IN_TIME_VAR]' AS p_union,
    person_studyat_organisation FOR SYSTEM_TIME AS OF '[POINT_IN_TIME_VAR]' AS o_union
WHERE o_union.person_id = p_union.id

-- b)
SELECT distinct m_union.id, p_union.id
FROM (SELECT id, creator_id FROM comment FOR SYSTEM_TIME AS OF '[POINT_IN_TIME_VAR]
    UNION
    SELECT id, creator_id FROM post FOR SYSTEM_TIME AS OF '[POINT_IN_TIME_VAR]') AS m_union,
    person FOR SYSTEM_TIME AS OF '[POINT_IN_TIME_VAR]' AS p_union
WHERE m_union.creator_id = p_union.id;

-- 2.3
-- a)
SELECT o_union.organisation_id, count(p_union.id) cnt
FROM person FOR SYSTEM_TIME AS OF '[POINT_IN_TIME_VAR]' AS p_union,
    person_studyat_organisation FOR SYSTEM_TIME AS OF '[POINT_IN_TIME_VAR]' AS o_union
WHERE o_union.person_id = p_union.id
GROUP BY o_union.organisation_id;

-- 3.3
-- a)
SELECT m_union.browserused, count(m_union.id)
FROM (SELECT id, browserused FROM comment FOR SYSTEM_TIME AS OF '[POINT_IN_TIME_VAR]
    UNION
    SELECT id, browserused FROM post FOR SYSTEM_TIME AS OF '[POINT_IN_TIME_VAR]') AS m_union
GROUP BY m_union.browserused;

-- b)
SELECT p_union.browserused, count(p_union.id)
FROM person FOR SYSTEM_TIME AS OF '[POINT_IN_TIME_VAR]' AS p_union
GROUP BY p_union.browserused;

-- c)
SELECT m_union.length, count(m_union.id)
FROM (SELECT id, length FROM comment FOR SYSTEM_TIME AS OF '[POINT_IN_TIME_VAR]'
    UNION
    SELECT id, length FROM post FOR SYSTEM_TIME AS OF '[POINT_IN_TIME_VAR]') AS m_union
WHERE m_union.length > [LENGTH_VARIABLE]
GROUP BY m_union.browserused;
```

Listing C.6: Queries Time Range, few temporal aspects with System d SQL.

```sql
UNION
SELECT id,length FROM post FOR SYSTEM_TIME AS OF 'POINT_IN_TIME_VAR'
AS m_union
GROUP BY m_union.length;

-- 1.1
-- a) SELECT distinct p_union.id
  FROM person FOR SYSTEM_TIME FROM min('POINT_IN_TIME_L_BOUND','[POINT_IN_TIME_U_BOUND]')
TO max('POINT_IN_TIME_L_BOUND','[POINT_IN_TIME_U_BOUND]') AS p_union
WHERE p_union.birthday = 'BIRTHDAY_VARIABLE';
-- b) SELECT distinct p_union.id
  FROM person FOR SYSTEM_TIME FROM min('POINT_IN_TIME_L_BOUND','[POINT_IN_TIME_U_BOUND]')
TO max('POINT_IN_TIME_L_BOUND','[POINT_IN_TIME_U_BOUND]') AS p_union
WHERE p_union.gender = 'GENDER';
-- c) SELECT distinct p_union.id
  FROM post FOR SYSTEM_TIME FROM min('POINT_IN_TIME_L_BOUND','[POINT_IN_TIME_U_BOUND]')
TO max('POINT_IN_TIME_L_BOUND','[POINT_IN_TIME_U_BOUND]') AS p_union
WHERE p_union.language = 'LANGUGAGE';
-- d) SELECT distinct p_union.id
  FROM person FOR SYSTEM_TIME FROM min('POINT_IN_TIME_L_BOUND','[POINT_IN_TIME_U_BOUND]')
TO max('POINT_IN_TIME_L_BOUND','[POINT_IN_TIME_U_BOUND]') AS p_union
WHERE p_union.birthday > 'BIRTHDAY_VARIABLE';
-- e) SELECT distinct m_union.id
  FROM (SELECT id,length FROM comment FOR SYSTEM_TIME FROM min('POINT_IN_TIME_L_BOUND','[POINT_IN_TIME_U_BOUND]')
  TO max('POINT_IN_TIME_L_BOUND','[POINT_IN_TIME_U_BOUND]')
  UNION
  SELECT id,length FROM post FOR SYSTEM_TIME FROM min('POINT_IN_TIME_L_BOUND','[POINT_IN_TIME_U_BOUND]')
  TO max('POINT_IN_TIME_L_BOUND','[POINT_IN_TIME_U_BOUND]'))
  AS m_union
WHERE m_union.length > LENGTH_VARIABLE;

-- 1.2
-- a) SELECT DISTINCT o_union.organisation_id
  FROM person FOR SYSTEM_TIME FROM min('POINT_IN_TIME_L_BOUND','[POINT_IN_TIME_U_BOUND]')
TO max('POINT_IN_TIME_L_BOUND','[POINT_IN_TIME_U_BOUND]') AS p_union,
  person_studyat_organisation FOR SYSTEM_TIME FROM min('POINT_IN_TIME_L_BOUND','[POINT_IN_TIME_U_BOUND]')
TO max('POINT_IN_TIME_L_BOUND','[POINT_IN_TIME_U_BOUND]') AS o_union
WHERE o_union.person_id = p_union.id
AND p_union.birthday = 'BIRTHDAY_VARIABLE';
-- b) SELECT DISTINCT o_union.organisation_id
```

---

UNION
SELECT id,length FROM post FOR SYSTEM_TIME AS OF 'POINT_IN_TIME_VAR'
AS m_union
GROUP BY m_union.length;
FROM person FOR SYSTEM_TIME FROM min('POINT_IN_TIME_L_BOUND'), 'POINT_IN_TIME_U_BOUND') TO max('POINT_IN_TIME_L_BOUND'), 'POINT_IN_TIME_U_BOUND') AS p_union,
person_studyat_organisation FOR SYSTEM_TIME FROM min('POINT_IN_TIME_L_BOUND'), 'POINT_IN_TIME_U_BOUND') TO max('POINT_IN_TIME_L_BOUND'), 'POINT_IN_TIME_U_BOUND') AS o_union
WHERE o_union.person_id = p_union.id
AND p_union.gender = 'GENDER';

-- c)
SELECT DISTINCT p_union.id
FROM post FOR SYSTEM_TIME FROM min('POINT_IN_TIME_L_BOUND'), 'POINT_IN_TIME_U_BOUND') TO max('POINT_IN_TIME_L_BOUND'), 'POINT_IN_TIME_U_BOUND') AS m_union,
person FOR SYSTEM_TIME FROM min('POINT_IN_TIME_L_BOUND'), 'POINT_IN_TIME_U_BOUND') TO max('POINT_IN_TIME_L_BOUND'), 'POINT_IN_TIME_U_BOUND') AS p_union
WHERE m_union.creator_id = p_union.id
AND m_union.language = 'LANGUAGE';

-- d)
SELECT DISTINCT p_union.id
FROM (SELECT id,creator_id
    FROM comment FOR SYSTEM_TIME FROM min('POINT_IN_TIME_L_BOUND'), 'POINT_IN_TIME_U_BOUND') TO max('POINT_IN_TIME_L_BOUND'), 'POINT_IN_TIME_U_BOUND')
WHERE length > LENGTH_VARIABLE
UNION
SELECT id,creator_id
    FROM post FOR SYSTEM_TIME FROM min('POINT_IN_TIME_L_BOUND'), 'POINT_IN_TIME_U_BOUND') TO max('POINT_IN_TIME_L_BOUND'), 'POINT_IN_TIME_U_BOUND')
WHERE length > LENGTH_VARIABLE
AS m_union,
person FOR SYSTEM_TIME FROM min('POINT_IN_TIME_L_BOUND'), 'POINT_IN_TIME_U_BOUND') TO max('POINT_IN_TIME_L_BOUND'), 'POINT_IN_TIME_U_BOUND') AS p_union
WHERE m_union.creator_id = p_union.id;

-- 1.3
-- a)
SELECT count( distinct p_union.id)
FROM person FOR SYSTEM_TIME FROM min('POINT_IN_TIME_L_BOUND'), 'POINT_IN_TIME_U_BOUND') TO max('POINT_IN_TIME_L_BOUND'), 'POINT_IN_TIME_U_BOUND') AS p_union
WHERE p_union.gender = 'GENDER';

-- b)
SELECT m_union.browserused, count(m_union.id)
FROM (SELECT id,length,browserused FROM comment FOR SYSTEM_TIME FROM min('POINT_IN_TIME_L_BOUND'), 'POINT_IN_TIME_U_BOUND') TO max('POINT_IN_TIME_L_BOUND'), 'POINT_IN_TIME_U_BOUND')
UNION
SELECT id,length,browserused FROM post FOR SYSTEM_TIME FROM min('POINT_IN_TIME_L_BOUND'), 'POINT_IN_TIME_U_BOUND') TO max('POINT_IN_TIME_L_BOUND'), 'POINT_IN_TIME_U_BOUND')
AS m_union
WHERE m_union.length > LENGTH_VARIABLE
GROUP BY m_union.browserused;

-- c)
SELECT m_union.length, count(m_union.id) cnt
FROM (SELECT id,length,browserused FROM comment FOR SYSTEM_TIME FROM min('POINT_IN_TIME_L_BOUND'), 'POINT_IN_TIME_U_BOUND') TO max('POINT_IN_TIME_L_BOUND'), 'POINT_IN_TIME_U_BOUND')
UNION
SELECT id,length,browserused FROM post FOR SYSTEM_TIME FROM min('POINT_IN_TIME_L_BOUND'), 'POINT_IN_TIME_U_BOUND') TO max('POINT_IN_TIME_L_BOUND'), 'POINT_IN_TIME_U_BOUND')
AS m_union
WHERE m_union.length > LENGTH_VARIABLE
GROUP BY m_union.length;
UNION
SELECT id,length,browserused FROM post FOR SYSTEM_TIME FROM min('[POINT_IN_TIME_L_BOUND]', '[POINT_IN_TIME_U_BOUND]') TO max('[POINT_IN_TIME_L_BOUND]', '[POINT_IN_TIME_U_BOUND]')
AS m_union
WHERE m_union.browserused = '[BROWSER_VARIABLE]' GROUP BY m_union.length;

-- 2.2
-- a)
SELECT p_union.id, o_union.organisation_id
FROM person FOR SYSTEM_TIME FROM min('[POINT_IN_TIME_L_BOUND]', '[POINT_IN_TIME_U_BOUND]') TO max('[POINT_IN_TIME_L_BOUND]', '[POINT_IN_TIME_U_BOUND]') AS p_union,
person_studyat_organisation FOR SYSTEM_TIME FROM min('[POINT_IN_TIME_L_BOUND]', '[POINT_IN_TIME_U_BOUND]') TO max('[POINT_IN_TIME_L_BOUND]', '[POINT_IN_TIME_U_BOUND]') AS o_union
WHERE o_union.person_id = p_union.id

-- b)
SELECT m_union.id, p_union.id
FROM (SELECT id,creator_id FROM comment FOR SYSTEM_TIME FROM min('[POINT_IN_TIME_L_BOUND]', '[POINT_IN_TIME_U_BOUND]') TO max('[POINT_IN_TIME_L_BOUND]', '[POINT_IN_TIME_U_BOUND]')
UNION
SELECT id,creator_id FROM post FOR SYSTEM_TIME FROM min('[POINT_IN_TIME_L_BOUND]', '[POINT_IN_TIME_U_BOUND]') TO max('[POINT_IN_TIME_L_BOUND]', '[POINT_IN_TIME_U_BOUND]'))
AS m_union,
person FOR SYSTEM_TIME FROM min('[POINT_IN_TIME_L_BOUND]', '[POINT_IN_TIME_U_BOUND]') TO max('[POINT_IN_TIME_L_BOUND]', '[POINT_IN_TIME_U_BOUND]') AS p_union
WHERE m_union.creator_id = p_union.id;

-- 2.3
-- a)
SELECT o_union.organisation_id, count(p_union.id) cnt
FROM person FOR SYSTEM_TIME FROM min('[POINT_IN_TIME_L_BOUND]', '[POINT_IN_TIME_U_BOUND]') TO max('[POINT_IN_TIME_L_BOUND]', '[POINT_IN_TIME_U_BOUND]') AS p_union,
person_studyat_organisation FOR SYSTEM_TIME FROM min('[POINT_IN_TIME_L_BOUND]', '[POINT_IN_TIME_U_BOUND]') TO max('[POINT_IN_TIME_L_BOUND]', '[POINT_IN_TIME_U_BOUND]') AS o_union
WHERE o_union.person_id = p_union.id GROUP BY o_union.organisation_id;

-- 3.3
-- a)
SELECT m_union.browserused, count(m_union.id)
FROM (SELECT id,browserused FROM comment FOR SYSTEM_TIME FROM min('[POINT_IN_TIME_L_BOUND]', '[POINT_IN_TIME_U_BOUND]') TO max('[POINT_IN_TIME_L_BOUND]', '[POINT_IN_TIME_U_BOUND]')
UNION
SELECT id,browserused FROM post FOR SYSTEM_TIME FROM min('[POINT_IN_TIME_L_BOUND]', '[POINT_IN_TIME_U_BOUND]') TO max('[POINT_IN_TIME_L_BOUND]', '[POINT_IN_TIME_U_BOUND]'))
AS m_union
GROUP BY m_union.browserused;

-- b)
SELECT p_union.browserused, count(p_union.id)
APPENDIX C. BENCHMARK QUERIES

FROM person FOR SYSTEM_TIME FROM min('POINT_IN_TIME_U_BOUND') TO max('POINT_IN_TIME_U_BOUND') AS p_union
GROUP BY p_union.browserused;

-- c)
SELECT m_union.length, count(m_union.id) FROM
(SELECT id,length FROM comment FOR SYSTEM_TIME FROM min('POINT_IN_TIME_U_BOUND') TO max('POINT_IN_TIME_U_BOUND'))
UNION
SELECT id,length FROM post FOR SYSTEM_TIME FROM min('POINT_IN_TIME_U_BOUND') TO max('POINT_IN_TIME_U_BOUND'))
AS m_union
GROUP BY m_union.length;

Listing C.7: Queries Non Temporal with System m.

-- 1.1
-- a)
SELECT distinct p_union.id FROM person('REQUEST_FLAGS'='ALLROWS') AS p_union
WHERE p_union.birthday = '[BIRTHDAY_VARIABLE]';

-- b)
SELECT distinct p_union.id FROM person('REQUEST_FLAGS'='ALLROWS') AS p_union
WHERE p_union.gender = '[GENDER]';

-- c)
SELECT distinct p_union.id FROM post('REQUEST_FLAGS'='ALLROWS') AS p_union
WHERE p_union.language = '[LANUGAGE]';

-- d)
SELECT distinct p_union.id FROM person('REQUEST_FLAGS'='ALLROWS') AS p_union
WHERE p_union.birthday > '[BIRTHDAY_VARIABLE]';

-- e)
SELECT distinct m_union.id FROM (SELECT id,length FROM comment('REQUEST_FLAGS'='ALLROWS')
UNION
SELECT id,length FROM post('REQUEST_FLAGS'='ALLROWS')) AS m_union
WHERE m_union.length > [LENGTH_VARIABLE];

-- 1.2
-- a)
SELECT distinct o_union.organisation_id FROM person('REQUEST_FLAGS'='ALLROWS') AS p_union,
person_studyat_organisation('REQUEST_FLAGS'='ALLROWS') AS o_union
WHERE o_union.person_id = p_union.id
AND p_union.birthday = '[BIRTHDAY_VARIABLE]';

-- b)
SELECT distinct o_union.organisation_id FROM person('REQUEST_FLAGS'='ALLROWS') AS p_union,
person_studyat_organisation('REQUEST_FLAGS'='ALLROWS') AS o_union
WHERE o_union.person_id = p_union.id
AND p_union.gender = '[GENDER]';
SELECT distinct p_union.id  
FROM post('REQUEST_FLAGS'='ALLROWS') AS m_union,  
person('REQUEST_FLAGS'='ALLROWS') AS p_union  
WHERE m_union.creator_id=p_union.id  
AND m_union.language = '[LANGUAGE]';

-- d)  
SELECT distinct p_union.id  
FROM (SELECT id,creator_id,length FROM comment('REQUEST_FLAGS'='ALLROWS')  
UNION  
SELECT id,creator_id,length FROM post('REQUEST_FLAGS'='ALLROWS')) AS m_union,  
person('REQUEST_FLAGS'='ALLROWS') AS p_union  
WHERE m_union.creator_id=p_union.id  
AND m_union.length > [LENGTH_VARIABLE];

-- 1.3
-- a)  
SELECT count( distinct p_union.id)  
FROM person('REQUEST_FLAGS'='ALLROWS') AS p_union  
WHERE p_union.gender = '[GENDER]';

-- b)  
SELECT m_union.browserused, count(m_union.id)  
FROM (SELECT id,length,browserused FROM comment('REQUEST_FLAGS'='ALLROWS')  
UNION  
SELECT id,length,browserused FROM post('REQUEST_FLAGS'='ALLROWS')) AS m_union  
WHERE m_union.length > [LENGTH_VARIABLE]  
GROUP BY m_union.browserused;

-- c)  
SELECT m_union.length, count(m_union.id) cnt  
FROM (SELECT id,length,browserused FROM comment('REQUEST_FLAGS'='ALLROWS')  
UNION  
SELECT id,length,browserused FROM post('REQUEST_FLAGS'='ALLROWS')) AS m_union  
WHERE m_union.browserused = '[BROWSER_VARIABLE]'  
GROUP BY m_union.length;

-- 2.2
-- a)  
SELECT distinct p_union.id, o_union.organisation_id  
FROM person('REQUEST_FLAGS'='ALLROWS') AS p_union,  
person_studyat_organisation('REQUEST_FLAGS'='ALLROWS') AS o_union  
WHERE o_union.person_id = p_union.id  
-- PRODUCED OOM

-- b)  
SELECT distinct m_union.id, p_union.id  
FROM (SELECT id,creator_id FROM comment('REQUEST_FLAGS'='ALLROWS')  
UNION  
SELECT id,creator_id FROM post('REQUEST_FLAGS'='ALLROWS')) AS m_union,  
person('REQUEST_FLAGS'='ALLROWS') AS p_union  
WHERE m_union.creator_id = p_union.id;

-- 2.3
-- a)  
SELECT o_union.organisation_id, count(p_union.id) cnt  
FROM person('REQUEST_FLAGS'='ALLROWS') AS p_union,  
person_studyat_organisation('REQUEST_FLAGS'='ALLROWS') AS o_union  
WHERE o_union.person_id = p_union.id  
GROUP BY o_union.organisation_id;
-- 3.3
-- a)
SELECT m_union.browserused, count(m_union.id)
FROM (SELECT id,browserused FROM comment('REQUEST_FLAGS'='ALLROWS')
UNION
SELECT id,browserused FROM post('REQUEST_FLAGS'='ALLROWS')) AS m_union
GROUP BY m_union.browserused;

-- b)
SELECT p_union.browserused, count(p_union.id)
FROM person('REQUEST_FLAGS'='ALLROWS') AS p_union
GROUP BY p_union.browserused;

-- c)
SELECT m_union.length, count(m_union.id)
FROM (SELECT id,length FROM comment('REQUEST_FLAGS'='ALLROWS')
UNION
SELECT id,length FROM post('REQUEST_FLAGS'='ALLROWS')) AS m_union
GROUP BY m_union.length;

Listing C.8: Queries Point in Time with System m.

-- 1.1
-- a)
SELECT distinct p_union.id
FROM person AS p_union
WHERE p_union.birthday = '[BIRTHDAY_VARIABLE]' AS OF utctimestamp '[POINT_IN_TIME_VAR]';

-- b)
SELECT distinct p_union.id
FROM person p_union
WHERE p_union.gender = '[GENDER]' AS OF utctimestamp '[POINT_IN_TIME_VAR]';

-- c)
SELECT distinct p_union.id
FROM post AS p_union
WHERE p_union.language = '[LANUGAGE]' AS OF utctimestamp '[POINT_IN_TIME_VAR]';

-- d)
SELECT distinct p_union.id
FROM person p_union
WHERE p_union.birthday > '[BIRTHDAY_VARIABLE]' AS OF utctimestamp '[POINT_IN_TIME_VAR]';

-- e)
SELECT distinct m_union.id
FROM (SELECT id,length FROM comment
UNION
SELECT id,length FROM post ) AS m_union
WHERE m_union.length > [LENGTH_VARIABLE] AS OF utctimestamp '[POINT_IN_TIME_VAR]';

-- 1.2
-- a)
SELECT distinct o_union.organisation_id
FROM person AS p_union,
person_studyat_organisation AS o_union
WHERE o_union.person_id = p_union.id
    AND p_union.birthday = '[BIRTHDAY_VARIABLE]'
AS OF utctimestamp '[POINT_IN_TIME_VAR]';

-- b) SELECT DISTINCT o_union.organisation_id
    FROM person AS p_union,
        person_studyat_organisation AS o_union
WHERE o_union.person_id = p_union.id
    AND p_union.gender = '[GENDER]'
AS OF utctimestamp '[POINT_IN_TIME_VAR]';

-- c) SELECT DISTINCT p_union.id
    FROM post AS m_union,
        person AS p_union
WHERE m_union.creator_id=p_union.id
    AND m_union.language = '[LANGUAGE]'
AS OF utctimestamp '[POINT_IN_TIME_VAR]';

-- d) SELECT DISTINCT p_union.id
    FROM (SELECT id,creator_id,length FROM comment
           UNION
           SELECT id,creator_id,length FROM post) AS m_union,
        person AS p_union
WHERE m_union.creator_id=p_union.id
    AND m_union.length > [LENGTH_VARIABLE]
AS OF utctimestamp '[POINT_IN_TIME_VAR]';

-- 1.3 -- a) SELECT count( distinct p_union.id)
    FROM person AS p_union
WHERE p_union.gender = '[GENDER]'
AS OF utctimestamp '[POINT_IN_TIME_VAR]';

-- b) SELECT m_union.browserused, count(m_union.id)
    FROM (SELECT id,length,browserused FROM comment
           UNION
           SELECT id,length,browserused FROM post) AS m_union
WHERE m_union.length > [LENGTH_VARIABLE]
GROUP BY m_union.browserused
AS OF utctimestamp '[POINT_IN_TIME_VAR]';

-- c) SELECT m_union.length, count(m_union.id) cnt
    FROM (SELECT id,length,browserused FROM comment
           UNION
           SELECT id,length,browserused FROM post) AS m_union
WHERE m_union.browserused = '[BROWSER_VARIABLE]'
GROUP BY m_union.length
AS OF utctimestamp '[POINT_IN_TIME_VAR]';

-- 2.2 -- a) SELECT distinct p_union.id, o_union.organisation_id
    FROM person AS p_union,
        person_studyat_organisation AS o_union
WHERE o_union.person_id = p_union.id
AS OF utctimestamp '[POINT_IN_TIME_VAR]';
APPENDIX C. BENCHMARK QUERIES

Listing C.9: Queries Time Range, few temporal aspects with System m.

-- b)
SELECT distinct m_union.id, p_union.id
FROM (SELECT id,creator_id FROM comment
     UNION
     SELECT id,creator_id FROM post) AS m_union,
     person AS p_union
WHERE m_union.creator_id = p_union.id
AS OF utctimestamp '[POINT_IN_TIME_VAR]';

-- 2.3
-- a)
SELECT o_union.organisation_id, count(p_union.id) cnt
FROM person AS p_union,
     person_studyat_organisation AS o_union
WHERE o_union.person_id = p_union.id
GROUP BY o_union.organisation_id
AS OF utctimestamp '[POINT_IN_TIME_VAR]';

-- 3.3
-- a)
SELECT m_union.browserused, count(m_union.id)
FROM (SELECT id,browserused FROM comment
     UNION
     SELECT id,browserused FROM post) AS m_union
GROUP BY m_union.browserused
AS OF utctimestamp '[POINT_IN_TIME_VAR]';

-- b)
SELECT p_union.browserused, count(p_union.id)
FROM person AS p_union
GROUP BY p_union.browserused
AS OF utctimestamp '[POINT_IN_TIME_VAR]';

-- c)
SELECT m_union.length, count(m_union.id)
FROM (SELECT id,length FROM comment
     UNION
     SELECT id,length FROM post) AS m_union
GROUP BY m_union.length
AS OF utctimestamp '[POINT_IN_TIME_VAR]';

-- 1.1
-- a)
SELECT distinct p_union.id
FROM person('REQUEST_FLAGS'='ALLROWS') AS p_union
WHERE p_union.birthday = '[BIRTHDAY_VARIABLE]' AND p_union."$validfrom$" <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE]
     OR p_union."$validto$" <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE]
     OR p_union."$validfrom$" >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE]
     OR p_union."$validto$" >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE]);

-- 1.1
-- b)
SELECT distinct p_union.id
FROM person('REQUEST_FLAGS'='ALLROWS') AS p_union
WHERE p_union.gender = '[GENDER]'
AND ( ([POINT_IN_TIME_L_BOUND] <= p_union."validfrom" AND p_union."validfrom" <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
    OR ([POINT_IN_TIME_L_BOUND] <= p_union."validto" AND p_union."validto" <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
    OR ([POINT_IN_TIME_L_BOUND] >= p_union."validfrom" AND p_union."validto" >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE]))
    )
)

-- c)
SELECT distinct p_union.id
FROM post('REQUEST_FLAGS'='ALLROWS') AS p_union
WHERE p_union.language = '[LANGUAGE]'
AND ( ([POINT_IN_TIME_L_BOUND] <= p_union."validfrom" AND p_union."validfrom" <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
    OR ([POINT_IN_TIME_L_BOUND] <= p_union."validto" AND p_union."validto" <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
    OR ([POINT_IN_TIME_L_BOUND] >= p_union."validfrom" AND p_union."validto" >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE]))
)

-- d)
SELECT distinct p_union.id
FROM person('REQUEST_FLAGS'='ALLROWS') AS p_union
WHERE p_union.birthday > '[BIRTHDAY_VARIABLE]'
AND ( ([POINT_IN_TIME_L_BOUND] <= p_union."validfrom" AND p_union."validfrom" <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
    OR ([POINT_IN_TIME_L_BOUND] <= p_union."validto" AND p_union."validto" <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
    OR ([POINT_IN_TIME_L_BOUND] >= p_union."validfrom" AND p_union."validto" >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE]))
)

-- e)
SELECT distinct m_union.id,
    m_union.length,
    m_union."validfrom", m_union."validto"
FROM (SELECT id,length,"validfrom","validto"
    FROM comment('REQUEST_FLAGS'='ALLROWS')
    UNION
    SELECT id,length,"validfrom","validto"
    FROM post('REQUEST_FLAGS'='ALLROWS')) AS m_union
WHERE m_union.length > [LENGTH_VARIABLE]
AND ( ([POINT_IN_TIME_L_BOUND] <= m_union."validfrom" AND m_union."validfrom" <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
    OR ([POINT_IN_TIME_L_BOUND] <= m_union."validto" AND m_union."validto" <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
    OR ([POINT_IN_TIME_L_BOUND] >= m_union."validfrom" AND m_union."validto" >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE]))
)

-- 1.2
-- a)
APPENDIX C. BENCHMARK QUERIES

-- b) 
SELECT DISTINCT o_union.organisation_id
FROM person('REQUEST_FLAGS'='ALLROWS') AS p_union,
    person_studyat_organisation('REQUEST_FLAGS'='ALLROWS') AS o_union
WHERE o_union.person_id = p_union.id
    AND o_union.$validfrom$ <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE]
    OR o_union.$validto$ <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE]
    OR (o_union.$validfrom$ <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE]
        AND o_union.$validto$ >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE]);

-- c) 
SELECT DISTINCT p_union.id
FROM post('REQUEST_FLAGS'='ALLROWS') AS m_union,
    person('REQUEST_FLAGS'='ALLROWS') AS p_union
WHERE m_union.creator_id = p_union.id
    AND m_union.$validfrom$ <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE]
    OR m_union.$validto$ <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE]
    OR (m_union.$validfrom$ <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE]
        AND m_union.$validto$ >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE]);

-- d) 
SELECT DISTINCT p_union.id
FROM (SELECT id,creator_id,length,$validfrom$,$validto$ FROM comment('REQUEST_FLAGS'='ALLROWS')) AS m_union
    UNION (SELECT id,creator_id,length,$validfrom$,$validto$ FROM post('REQUEST_FLAGS'='ALLROWS')) AS m_union
WHERE m_union.creator_id = p_union.id
    AND m_union.length > [LENGTH_VARIABLE]
    OR (m_union.$validfrom$ <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
    OR (m_union.$validto$ <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE]);
OR (\([\text{POINT\_IN\_TIME\_L\_BOUND}]\geq p\_union.\"validfrom\"\) AND \(p\_union.\"validto\"\leq \([\text{POINT\_IN\_TIME\_L\_BOUND}]+[\text{DIFFERENCE}]\))
)

AND ( (\([\text{POINT\_IN\_TIME\_L\_BOUND}]\leq m\_union.\"validfrom\"\) AND \(m\_union.\"validfrom\"\leq \([\text{POINT\_IN\_TIME\_L\_BOUND}]+[\text{DIFFERENCE}]\))
OR (\([\text{POINT\_IN\_TIME\_L\_BOUND}]\leq m\_union.\"validto\"\) AND \(m\_union.\"validto\"\leq \([\text{POINT\_IN\_TIME\_L\_BOUND}]+[\text{DIFFERENCE}]\))
OR (\([\text{POINT\_IN\_TIME\_L\_BOUND}]\geq m\_union.\"validfrom\"\) AND \(m\_union.\"validto\"\geq \([\text{POINT\_IN\_TIME\_L\_BOUND}]+[\text{DIFFERENCE}]\))
)
)

-- 1.3
-- a)
SELECT count( distinct p\_union.id)
FROM person('REQUEST\_FLAGS'=\'ALLROWS\') AS p\_union
WHERE p\_union.gender = \'(\[GENDER]\)'
AND ( (\([\text{POINT\_IN\_TIME\_L\_BOUND}]\leq p\_union.\"validfrom\"\) AND p\_union.\"validfrom\"\leq \([\text{POINT\_IN\_TIME\_L\_BOUND}]+[\text{DIFFERENCE}]\))
OR (\([\text{POINT\_IN\_TIME\_L\_BOUND}]\leq p\_union.\"validto\"\) AND p\_union.\"validto\"\leq \([\text{POINT\_IN\_TIME\_L\_BOUND}]+[\text{DIFFERENCE}]\))
OR (\([\text{POINT\_IN\_TIME\_L\_BOUND}]\geq p\_union.\"validfrom\"\) AND p\_union.\"validto\"\geq \([\text{POINT\_IN\_TIME\_L\_BOUND}]+[\text{DIFFERENCE}]\))
)

-- b)
SELECT m\_union.browserused, count(m\_union.id)
FROM (SELECT id,length,browserused,\"validfrom\",\"validto\" FROM comment(' REQUEST\_FLAGS'=\'ALLROWS\') UNION
SELECT id,length,browserused,\"validfrom\",\"validto\" FROM post(' REQUEST\_FLAGS'=\'ALLROWS\'))) AS m\_union
WHERE m\_union.length > \[LENGTH\_VARIABLE\]
AND ( (\([\text{POINT\_IN\_TIME\_L\_BOUND}]\leq m\_union.\"validfrom\"\) AND m\_union.\"validfrom\"\leq \([\text{POINT\_IN\_TIME\_L\_BOUND}]+[\text{DIFFERENCE}]\))
OR (\([\text{POINT\_IN\_TIME\_L\_BOUND}]\leq m\_union.\"validto\"\) AND m\_union.\"validto\"\leq \([\text{POINT\_IN\_TIME\_L\_BOUND}]+[\text{DIFFERENCE}]\))
OR (\([\text{POINT\_IN\_TIME\_L\_BOUND}]\geq m\_union.\"validfrom\"\) AND m\_union.\"validto\"\geq \([\text{POINT\_IN\_TIME\_L\_BOUND}]+[\text{DIFFERENCE}]\))
)
GROUP BY m\_union.browserused;

-- c)
SELECT m\_union.length, count(m\_union.id) cnt
FROM (SELECT id,length,browserused,\"validfrom\",\"validto\" FROM comment(' REQUEST\_FLAGS'=\'ALLROWS\') UNION
SELECT id,length,browserused,\"validfrom\",\"validto\" FROM post(' REQUEST\_FLAGS'=\'ALLROWS\'))) AS m\_union
WHERE m\_union.browserused = \'(\[BROWSER\_VARIABLE]\)'
AND ( (\([\text{POINT\_IN\_TIME\_L\_BOUND}]\leq m\_union.\"validfrom\"\) AND m\_union.\"validfrom\"\leq \([\text{POINT\_IN\_TIME\_L\_BOUND}]+[\text{DIFFERENCE}]\))
OR (\([\text{POINT\_IN\_TIME\_L\_BOUND}]\leq m\_union.\"validto\"\) AND m\_union.\"validto\"\leq \([\text{POINT\_IN\_TIME\_L\_BOUND}]+[\text{DIFFERENCE}]\))
OR (\([\text{POINT\_IN\_TIME\_L\_BOUND}]\geq m\_union.\"validfrom\"\) AND m\_union.\"validto\"\geq \([\text{POINT\_IN\_TIME\_L\_BOUND}]+[\text{DIFFERENCE}]\))
)
GROUP BY m\_union.length;

-- 2.2
-- a)
SELECT p\_union.id, o\_union.organisation_id
FROM person('REQUEST\_FLAGS'=\'ALLROWS\') AS p\_union,
person_studyat_organisation('REQUEST_FLAGS'='ALLROWS') AS o_union
WHERE o_union.person_id = p_union.id
AND ( ([POINT_IN_TIME_L_BOUND] <= o_union."$validfrom$" AND o_union."$validfrom$" < [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
OR ([POINT_IN_TIME_L_BOUND] <= o_union."$validto$" AND o_union."$validto$" <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
OR ([POINT_IN_TIME_L_BOUND] >= o_union."$validfrom$" AND o_union."$validto$" >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
);

-- b)
SELECT m_union.id, p_union.id
FROM (SELECT id,creator_id,"$validfrom$","$validto$" FROM comment('REQUEST_FLAGS'='ALLROWS')) AS m_union,
person('REQUEST_FLAGS'='ALLROWS') AS p_union
WHERE m_union.creator_id = p_union.id
AND ( ([POINT_IN_TIME_L_BOUND] <= m_union."$validfrom$" AND m_union."$validfrom$" < [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
OR ([POINT_IN_TIME_L_BOUND] <= m_union."$validto$" AND m_union."$validto$" <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
OR ([POINT_IN_TIME_L_BOUND] >= m_union."$validfrom$" AND m_union."$validto$" >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
);

-- 2.3
-- a)
SELECT o_union.organisation_id, count(p_union.id) cnt
FROM person('REQUEST_FLAGS'='ALLROWS') AS p_union,
person_studyat_organisation('REQUEST_FLAGS'='ALLROWS') AS o_union
WHERE o_union.person_id = p_union.id
AND ( ([POINT_IN_TIME_L_BOUND] <= o_union."$validfrom$" AND o_union."$validfrom$" < [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
OR ([POINT_IN_TIME_L_BOUND] <= o_union."$validto$" AND o_union."$validto$" <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
OR ([POINT_IN_TIME_L_BOUND] >= o_union."$validfrom$" AND o_union."$validto$" >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
)
GROUP BY o_union.organisation_id;

-- 3.3
-- a)
SELECT m_union.browserused, count(m_union.id)
FROM (SELECT id,browserused,"$validfrom$","$validto$" FROM comment('REQUEST_FLAGS'='ALLROWS')) AS m_union,
(SELECT id,browserused,"$validfrom$","$validto$" FROM post('REQUEST_FLAGS'='ALLROWS')) AS m_union
WHERE ( ([POINT_IN_TIME_L_BOUND] <= m_union."$validfrom$" AND m_union."$validfrom$" < [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
OR ([POINT_IN_TIME_L_BOUND] <= m_union."$validto$" AND m_union."$validto$" <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
OR ([POINT_IN_TIME_L_BOUND] >= m_union."$validfrom$" AND m_union."$validto$" >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
)
GROUP BY m_union.browserused;

--b)
SELECT p_union.browserused, count(p_union.id)
FROM person('REQUEST_FLAGS'='ALLROWS') AS p_union
WHERE ( ([POINT_IN_TIME_L_BOUND] <= p_union."validfrom" AND p_union."validfrom" <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
  OR ([POINT_IN_TIME_L_BOUND] <= p_union."validto" AND p_union."validto" <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
  OR ([POINT_IN_TIME_L_BOUND] >= p_union."validfrom" AND p_union."validto" >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
)
GROUP BY p_union.browserused;

-- c)
SELECT m_union.length, count(m_union.id)
FROM (SELECT id,length,"validfrom","validto" FROM comment('REQUEST_FLAGS'='ALLROWS')

  UNION
  SELECT id,length,"validfrom","validto" FROM post('REQUEST_FLAGS'='ALLROWS')

WHERE ( ([POINT_IN_TIME_L_BOUND] <= m_union."validfrom" AND m_union."validfrom" <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
  OR ([POINT_IN_TIME_L_BOUND] <= m_union."validto" AND m_union."validto" <= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
  OR ([POINT_IN_TIME_L_BOUND] >= m_union."validfrom" AND m_union."validto" >= [POINT_IN_TIME_L_BOUND] + [DIFFERENCE])
)
GROUP BY m_union.length;
Eigenständigkeitserklärung


Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

Temporal Graph Data Management for in-memory Database Systems

Verfasst von (in Druckschrift):

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):

Rohr

Vorname(n):

Philipp Oliver

Ich bestätige mit meiner Unterschrift:

− Ich habe keine im Merkblatt „Zitier-Knigge“ beschriebene Form des Plagiats begangen.
− Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
− Ich habe keine Daten manipuliert.
− Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum

Starrkirch-Wil, 22. Februar 2015

Unterschrift(en)