Universität Leipzig

Institut für Informatik
Fakultät für Mathematik und Informatik
Abteilung Datenbanken

Distributed Graph Pattern Matching on Evolving Graphs

Masterarbeit

vorgelegt von:
Lukas Christ

Matrikelnummer:
3756213

Betreuer:
Kevin Gomez, M.Sc.
Prüfer:
Prof. Dr. Erhard Rahm
Dr. Eric Peukert

(page intentionally left blank)

(page intentionally left blank)

Abstract

Gradoop, the distributed graph processing system developed by the Database Group at Leipzig University, is able to handle two types of graphs: extended property graphs as defined by the Extended Property Graph Model (EPGM) [32] and temporal property graphs as defined by the Temporal Property Graph Model (TPGM) [57]. The main difference between the TPGM and the EPGM is that TPGM graphs contain temporal information and can be said to model evolving graphs. A common task in analyzing graphs is Pattern Matching, i.e. searching for occurrences of a pattern (query) in the graph. In Gradoop, a Pattern Matching operator is implemented for EPGM in Gradoop([28]), but not for TPGM graphs yet. The core idea of this implementation is to employ relational query planning in order to compute all matches of a pattern. This thesis aims at realizing pattern matching for TPGM graphs in Gradoop. First, the query language GDL is extended to support constraints on transaction and valid intervals. It is now possible to refer to them (and their boundaries) in a query and specify temporal relations in order to compare them among each other. Furthermore, the user is now able to compare interval durations as well as create custom intervals to compare them to transaction and valid intervals. Second, the existing Gradoop pattern matching implementation is adapted to support these new types of queries. Besides that, the pattern matching implementation is optimized via query rewriting and constraint selectivity estimation. The latter improves the query planning approach employed for processing queries while query rewriting aims at obtaining a more efficient query. Moreover, the TPGM pattern matching operator's performance for two different data sets is evaluated on a distributed cluster.

Lukas Christ
3756213

Contents

Lukas Christ
3756213

List of Figures

List of Tables

Listings

List of Algorithms

1. Introduction

Many real-world information can be viewed as a set of entities, possibly of different types, and links connecting these entities. Such information can be modelled as a graph, which is a set of *vertices* linked by *edges*. There are plenty of examples for graph data, among them the internet (URLs that contain links to other URLs), transportation networks (stations connected by e.g. trains) or *knowledge graphs*, e.g. in social networks. Figure 1.1 shows a small *knowledge (or data) graph* that could be taken from a social network like XING.

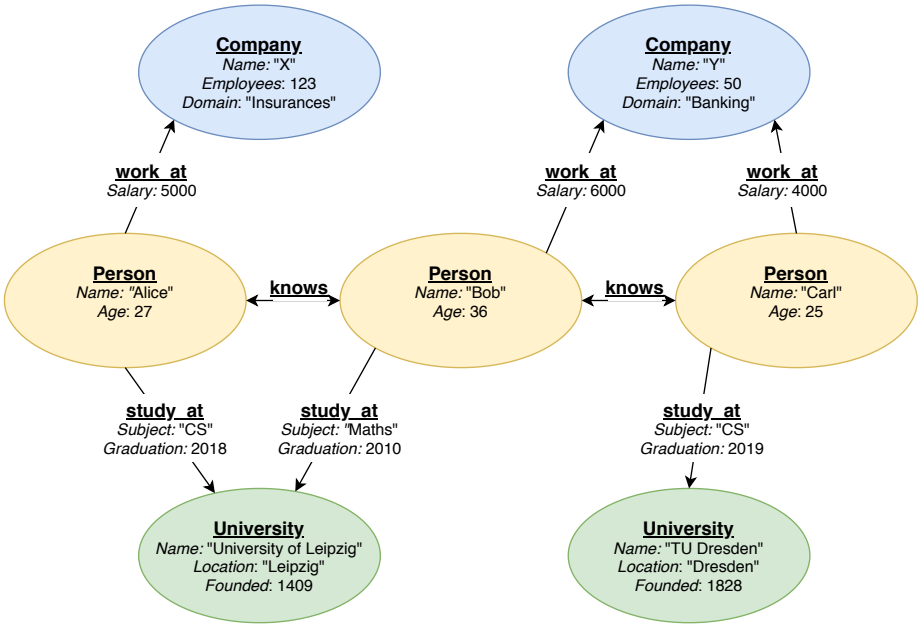

Figure 1.1.: Exemplary knowledge graph (property graph)

The vertices in this knowledge graph represent universities, persons and companies. They are linked via edges of different types, e.g. persons may know each other. Moreover, vertices and edges may contain further information, e.g. the name of a person. The graph depicted in Figure 1.1 is thus a *property graph*, which will be defined formally later (see Section 2.2).

In a lot of applications today, data graphs *evolve*, meaning that their structure and contained information changes over time. To give an example for evolving graphs, social network graphs (e.g. the facebook graph) typically grow rapidly due to user interactions, new posts etc. Furthermore, such graphs are usually very large. For example, social networks like facebook have hundreds of millions of active users.

Analyzing large evolving graphs can thus be considered a big data problem. One system that allows the analysis of large evolving graphs is Gradoop. Gradoop is developed at the University of Leipzig and provides several methods to investigate (evolving) graphs in parallel, so that also very large graphs can be processed in reasonable time.

There are many methods to analyse graphs theoretically. *Pattern Matching* is one of them. In Pattern Matching, the graph is searched for all occurrences of an interesting pattern. Patterns are also referred to as *queries*. For the example graph in Figure 1.1, it might be interesting to find

all pairs of persons that studied at the same university and work for the same company now. The corresponding pattern would be a graph containing two *Person* vertices both having edges to the same *University* and *Company* vertex. Pattern Matching will be defined formally in Section 2.4.

Gradoop already implements a Pattern Matching functionality, but only for static, i.e. non-evolving, graphs. However, for many types of queries, temporal aspects are relevant. To take up again the pattern example above, one might only be interested in persons that studied at the same university at the same time. Such temporal constraints on patterns cannot be handled by Gradoop's Pattern Matching component as of now.

Hence, the aim of this thesis is to implement Pattern Matching on evolving graphs (also referred to as *Temporal Pattern Matching*) in Gradoop.

The thesis is structured as follows: Section 2 is a more detailed and formal introduction to temporal or time-dependent Graph Pattern Matching in Gradoop. Section 3 provides an overview over related work. The new types of queries made possible by this thesis are presented in Section 4. A detailed discussion of the implementation in Gradoop is to be found in Section 5, followed by the description of optimization approaches in Section 6. The implemented program is evaluated in Section 7. Section 8 concludes the thesis with a brief discussion.

2. Foundations

In the following, the prerequisites for Time-dependent Graph Pattern Matching in Gradoop are introduced.

Section 2.1 provides the fundamental notations and definitions used throughout the thesis. Then, several models for data graphs are defined (2.2). After that, the graph processing system Gradoop is introduced (2.3), followed by the formal definition of (time-dependent) Graph Pattern Matching (2.4).

2.1. Prerequisites

Graphs are a core concept of Computer Science and of vital importance in this thesis.

Definition 1 (Graph)**.** *A* Graph *is a tuple* (V, E) *where* V *is a set of* vertices *and* $E \subseteq V \times V$ *a set of* edges *connecting two vertices.*

The terms *vertex* and *node* are used interchangeably. *Graph Element* is a hypernym used to refer to vertices and edges alike. E.g., the set $V \cup E$ contains all the graph elements of a graph (V, E).

If not stated otherwise, graphs are assumed to be directed, meaning that edges have a direction.

Definition 2 (Directed Graph)**.** *A graph* (V, E) *is said to be directed, if and only if* $(u, v) \in E \not\Rightarrow (v, u) \in E \ \forall \ u, v \in V, \ u \neq v$.

For an edge $e = (u, v)$, u is called the *source* or *source vertex/node* of e, while the terms *target* or *target vertex/node* refer to v. Unless stated otherwise, *self-loops*, i.e. edges whose source and target are identical, are allowed in a graph.

A *path* connects two vertices via a sequence of edges.

Definition 3 (Path)**.** *Let* $e_1 = (u_1, v_1)$, $e_2 = (u_2, v_2)$, ..., $e_n = (u_n, v_n)$ *be a sequence of edges with* $v_i = u_{i+1} \ \forall i (1 \leq i \leq (n-1))$. *Then,* e_1,...,e_n *is a* path *from* u_1 *to* v_n *with length* n.

Especially for the problem of Pattern Matching, a notion of *subgraphs* is needed.

Definition 4 (Subgraph)**.** *A graph* $G' = (V', E')$ *is a subgraph of another graph* $G = (V, E)$*, if and only if* $V' \subseteq V$ *and* $E' \subseteq ((V' \times V') \cap E)$. *The subgraph relation is noted* $G' \sqsubseteq G$. *A subgraph can be* induced *by a set of vertices* $V' \subseteq V$. *The subgraph induced by* $V' \subseteq V$ *is noted* $G[V']$ *and defined as* $(V', \{(u, v) \in E | u \in V', \ v \in V'\})$. *Analogously, a subgraph can be induced by a set of edges.*

2.2. Graph Data Models

Knowledge graphs, also called data graphs, are graphs in which vertices and edges carry information. The graph in Figure 1.1 is an example for a data graph. Such graphs have a wide range of applications, as pointed out in Section 1. Graph data models formalize the notion of knowledge graphs.

2.2.1. Property Graph Model

The arguably most prominent and simple graph data model is the Property Graph Model (PGM). For example, the graph database Neo4J[1] is based on this model [16]. In the PGM, vertices and edges have labels and may contain properties. Labels are arbitrary character sequences, while properties can be thought of as key-value pairs.

Definition 5 (Property Graph)**.** *A* Property Graph *is a 7-tuple* $(V, E, T, \tau, K, A, \kappa)$*.*

(V, E) *is a graph,* T *the set of* type labels*. To each graph element, a type label is assigned via the function* $\tau : V \cup E \to T$*.* K *and* A *denote the set of* property keys *and* property values*, respectively. The function* $\kappa : (V \cup E) \times K \to A \cup \{\epsilon\}$ *assigns properties as key-value pairs to graph elements.*

It is clear from the definition that not every graph element has a value for each possible property key, i.e. that properties are optional.

Figure 1.1 is an example for a Property Graph.

For the graph in Figure 1.1, $T = \{$*Company, Person, University, work_at, study_at*$\}$ and $K = \{$*Name, Employees, Domain, Age, ...*$\}$. Attributes can be of different types. In the example, there are string attributes like *Name* and integer attributes, e.g. *Age*.

The subgraph relation introduced in Definition 4 can be extended to Property Graphs in a straightforward way.

Definition 6 (Subgraph for Property Graphs)**.** *A property graph* $G' = (V', E', T, \tau', K, A, \kappa')$ *is a subgraph of another property graph* $G = (V, E, T, \tau, K, A, \kappa)$*, iff* $(V', E') \sqsubseteq (V, E)$*,* $\tau' = \tau_{|(V' \cup E')}$ *and* $\kappa = \kappa_{|(V' \cup E')}$*.*

Moreover, the notion of induced subgraphs given in Definition 4 is easily generalized for property graphs.

[1] `neo4j.com`, accessed August 2020

2.2.2. Extended Property Graph Model

Junghanns et al. [32, 28] propose an extension of the classic PGM, called *Extended Property Graph Model (EPGM)*.

Definition 7 (Extended Property Graph)**.** *An* Extended Property Graph *is a 9-tuple* $(L, l, V, E, T, \tau, K, A, \kappa)$*, where* $(V, E, T, \tau, K, A, \kappa)$ *is a Property Graph as defined in Definition 5.* L *is a set of* Logical Graphs*, which are subgraphs of* $(V, E, T, \tau, K, A, \kappa)$*. The function* $l : V \cup E \rightarrow \mathcal{P}(L) \backslash \emptyset$ *maps every graph element to the set of logical graphs in which it is contained.*

A set of Logical Graphs is also called a *Graph Collection*. Figure 2.1 shows an Extended Property Graph based on the Property Graph in Figure 1.1.

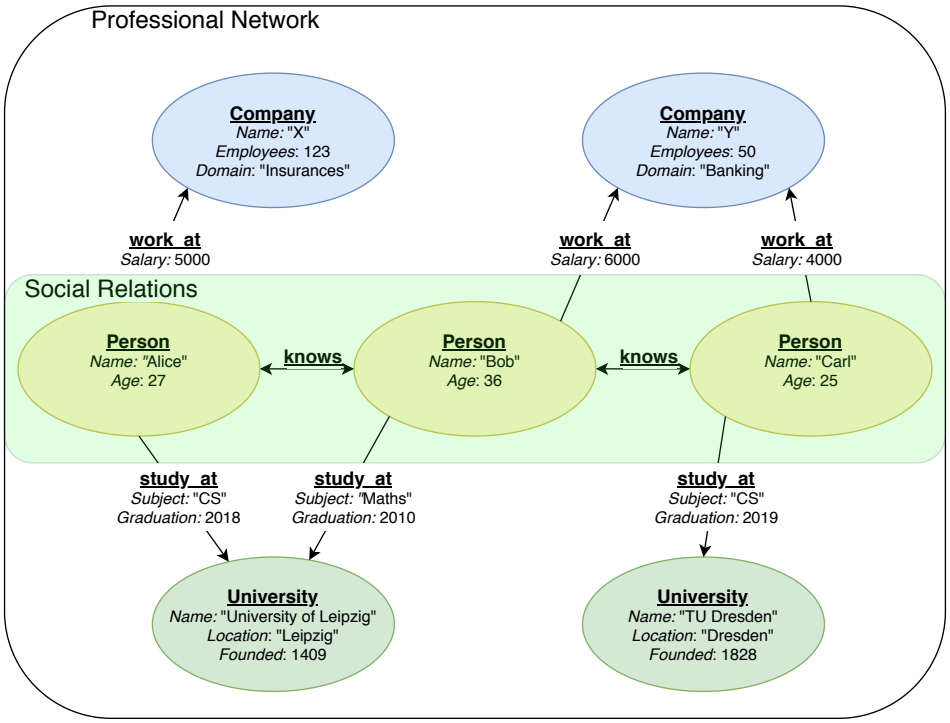

Figure 2.1.: Exemplary EPGM graph

The vertices and edges as well as their labels and properties of Figure 1.1 are left unchanged. However, now there are two logical graphs: the whole graph called *Professional Network* and the *Social Relations* subgraph containing only persons and interpersonal relationships. Logical graphs may overlap, as do the two logical graphs in the example. They can be considered views on the Property Graph.

Another addition to the Property Graph Model defined by EPGM is a set of operators on graphs, as listed in [32]. These operators include Pattern Matching, Subgraph Extraction, Filtering of Vertices and Edges and many more.

2.2.3. Temporal Property Graph Model

Neither PGM nor EPGM explicitly define temporal information for vertices and edges. This is different in the Temporal Property Graph Model (TPGM) proposed by Rost et al. [57]. In the TPGM, every vertex and edge is associated with two intervals: transaction and valid interval. The *transaction interval* denotes the period during which a graph element is held in the database. The *valid interval*, however, expresses semantic information about the element. It denotes the period during which an element is considered "true". Thus, the interpretation of the valid interval's meaning is context-dependent.

Definition 8 (Temporal Property Graph)**.** *A* Temporal Property Graph *is an Extended Property Graph in which every graph element has the four additional properties* tx-from*,* tx-to*,* val-from *and* val-to*. Their values are either time stamps or* ϵ*.*

The four extra properties represent the transaction interval ($[tx\text{-}from, tx\text{-}to]$) and the valid interval ($[val\text{-}from, val\text{-}to]$). Note that the definition allows these properties to be empty. Hence, intervals without a specified beginning or end are possible, e.g. when beginning or end are not known or intended. Furthermore, by omitting the *from* as well as the *to* time stamp, i.e. setting the corresponding property to ϵ, it can be expressed that the element does not have the respective interval. Because of that, graph elements having one or both of these intervals can coexist with non-temporal graph elements in the same database. Figure 2.2 shows a TPGM graph similar to Figure 2.1.

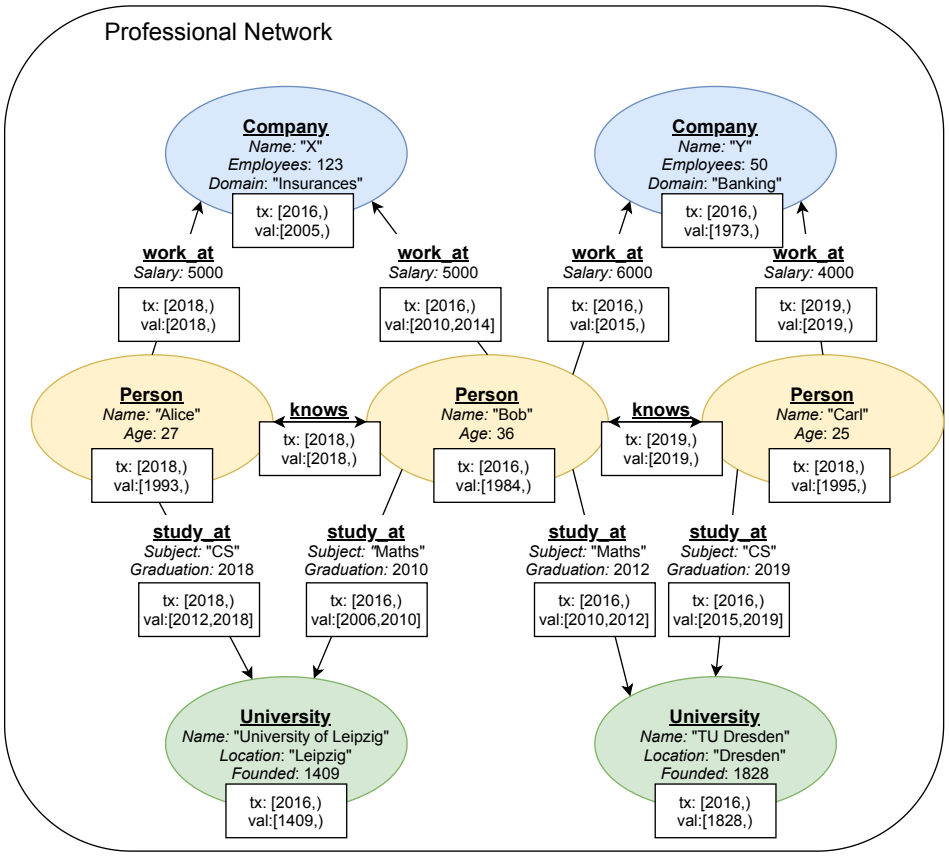

Figure 2.2.: Exemplary TPGM graph

The logical graph *Social Relations* was dropped for simplicity. Every vertex and edge in Figure 2.2 has a transaction and a valid interval. In the Figure and throughout this thesis, the intervals are often abbreviated *tx* and *val*. Both intervals are defined via the time stamps representing their beginning and end (both stamps are part of the interval). Interval boundaries are given in years here in order to keep the diagram clear. Of course, in TPGM implementations, the boundaries are more likely to be given in miliseconds since the UNIX epoch. All transaction intervals in the example begin in 2016 or later, indicating that the database was created in 2016. Furthermore, no transaction interval has an end time stamp yet so that all vertices and edges are present in the database as of now. Similarly, no vertex's valid interval has ended yet. The valid intervals of vertices in the example start with the founding year for universities and year of birth for Persons. Valid intervals for companies can also be interpreted as the companies' "lifetimes", e.g. company X may exist since 2005. The edges' valid intervals always start with the beginning of the relation described by the edge, e.g., Alice knows Bob since 2018. All *study_at* edges have closed valid intervals, as all three Persons already finished their studies. To give an example, since Alice graduated in 2018, the fact *Alice studies at University of Leipzig* can not be considered true anymore after 2018. Other than in Figure 2.1, Bob now has two *study_at* and *work_at* edges, with their valid intervals showing that he first studied at University of Leipzig from 2006 to 2010 and then went on to study at TU Dresden from 2010 to 2012. Analogously, he worked at company X from 2010 to 2014 and after that at company Y from 2015 on. It is clear that the facts in the graph of Figure 2.2 differ over time. Stated differently, the graph is evolving.

In the following Figures 2.3, 2.4 and 2.5, the valid facts of the TPGM graph in Figure 2.2 are shown as of 2008, 2016 and 2020 respectively.

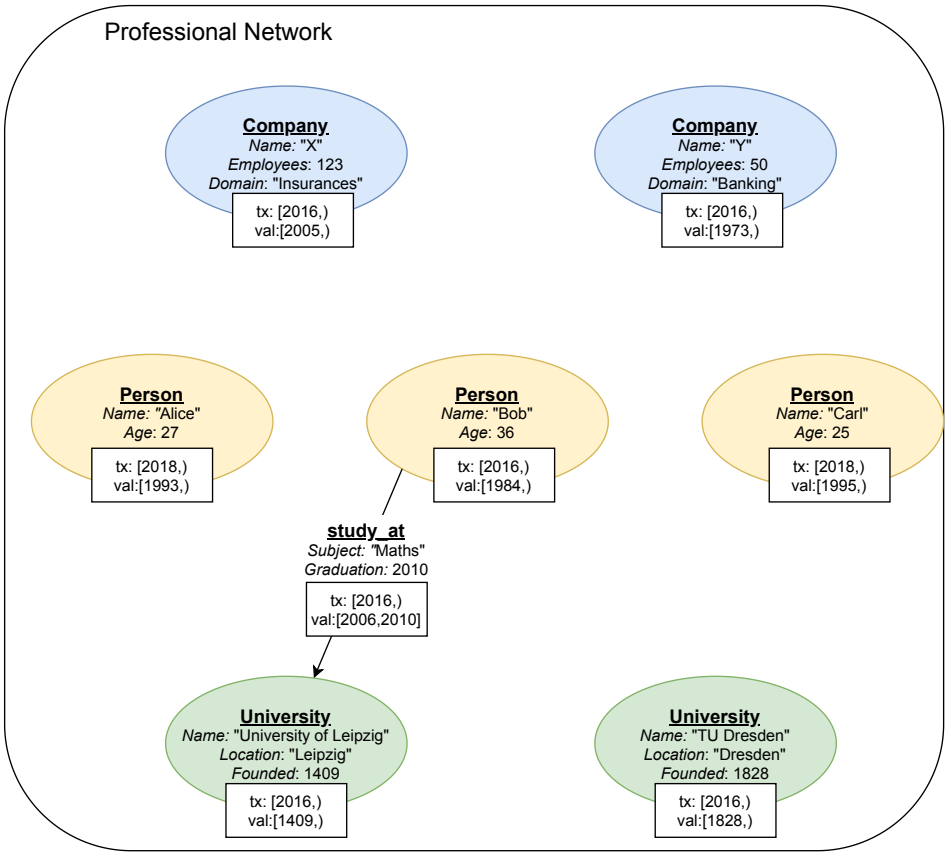

Figure 2.3.: Exemplary TPGM graph as of 2008

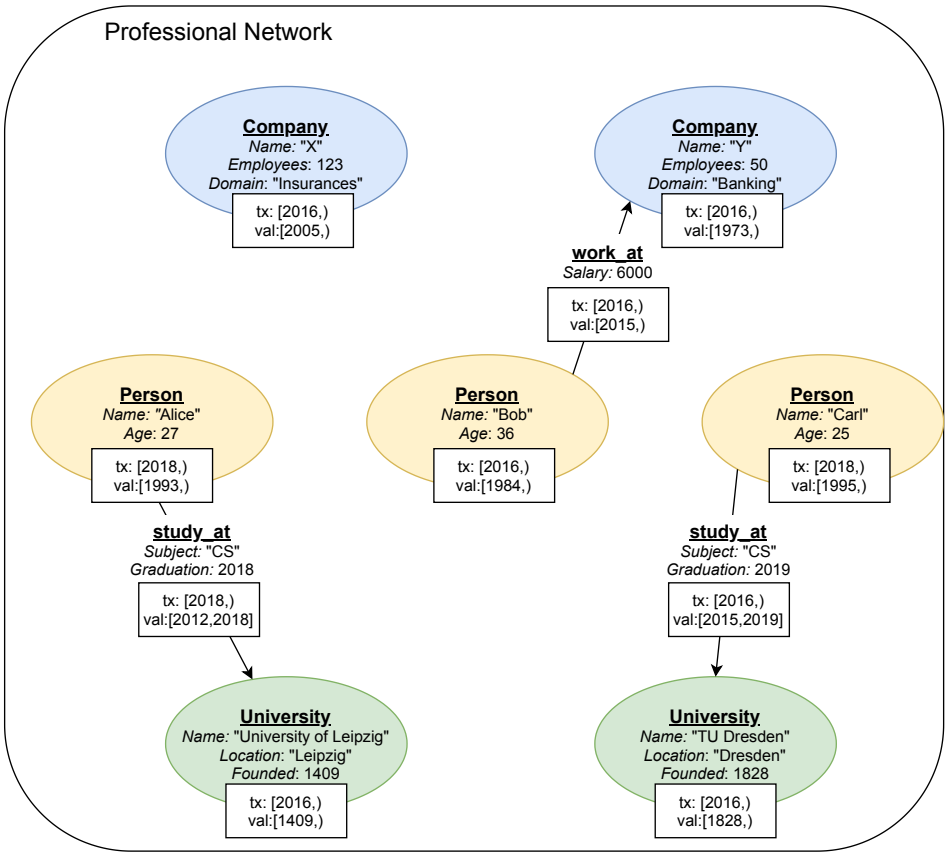

Figure 2.4.: Exemplary TPGM graph as of 2016

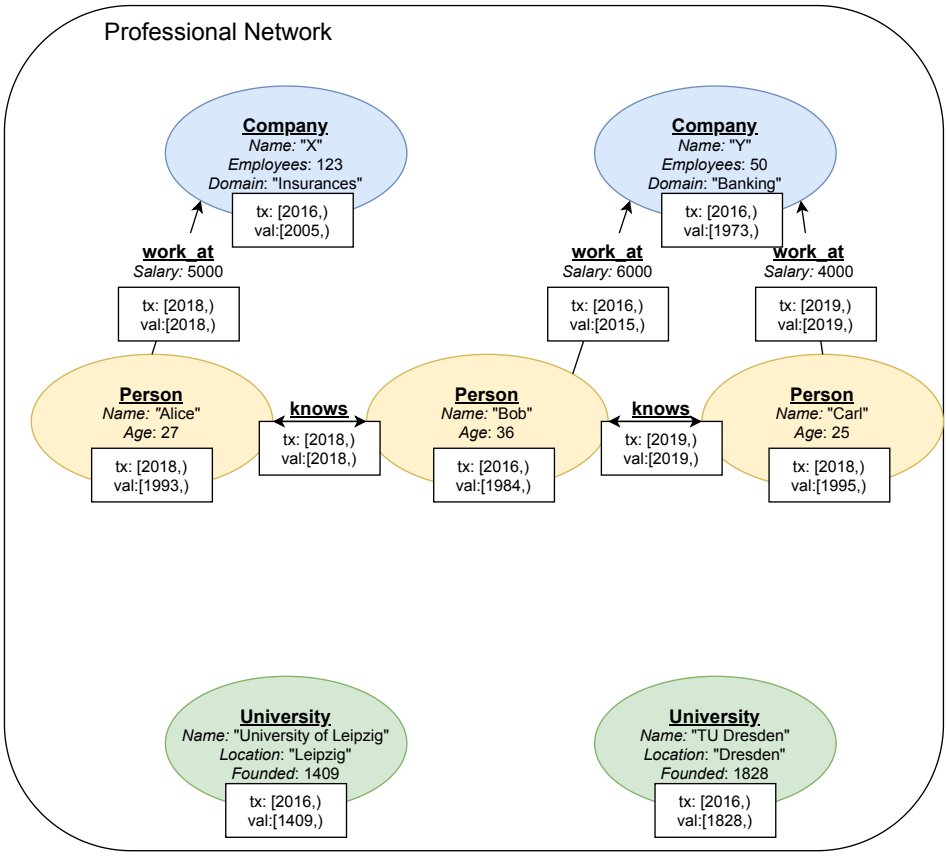

Figure 2.5.: Exemplary TPGM graph as of 2020

TPGM provides a suitable model for analysis of evolving knowledge graphs. Similar to EPGM, it defines further operators. These operators are listed in Table 2.1. For more details on the operators other than Pattern Matching, see Rost et al. [57].

Operator	Type	Description
Transformation	Graph → Graph	structure-preserving function application
Subgraph	Graph → Graph	filter subgraph via functions
Snapshot	Graph → Graph	graph as of a certain point in time or during an interval
Difference	Graph → Graph	difference of two snapshots
Grouping	Graph → GraphCollection	group vertices and edges
Pattern Matching	Graph → GraphCollection	find matches for a pattern

Table 2.1.: TPGM operators

In the following, Gradoop, the reference implementation of EPGM and TPGM, is presented.

2.3. Gradoop

Gradoop [32, 29] is a distributed graph processing system that is developed by the Database Group at the Leipzig University. It implements the EPGM including its operators and, at least partly, the TPGM: Temporal Property Graphs can already be loaded, however, not all operators defined by

TPGM (see Table 2.1) are already implemented. E.g., the Temporal Pattern Matching Operator is still missing.

In addition to the EPGM and TPGM operators, various graph analysis algorithms, e.g. PageRank, are provided.

Gradoop is an end-to-end system. Several methods to construct graphs from data are available. For example, graphs to load can be given in JSON or CSV. Analogously, different sinks are available to save transformed graphs.

Gradoop is capable of processing very large graphs due to its distributed architecture. To realize this architecture, Apache Flink[2] is employed. Flink is a distributed processing framework. It provides high-level functions to handle large amounts of data, so that a programmer need not care about the details of managing the distributed processing. There are two APIs in Flink, the DataStream API and the DataSet API. Both of them enable transforming a collection of elements. Available transformations are, among others, filtering, mapping, reducing and joining. The DataStream API is used for handling potentially unbounded streams, while the DataSetAPI is applicable to finite batches of data. Hence, Flink's DataSet API is well-suited for the distributed processing of (graph) database content and thus used in Gradoop.

The user can execute analytical workflows in Gradoop using the declarative GrALa (Graph Analytical Language), that is provided as a Java API. GrALa allows to specify pipelines combining the available graph operators and algorithms. All the transformations specified by the user are mapped to Flink pipelines internally.

As of now, Gradoop is work in progress and open-source[3].

2.4. Pattern Matching

A common task in the analysis of data graphs is Pattern Matching, i.e. finding *matches* for a certain *query (pattern)* in the given data graph. The following is devoted to formalize this rather vague intuition of Pattern Matching. The definitions given here are similar to those by Junghanns et al. [28], but differ in some details.

Definition 9 (Query/Pattern)**.** *A* query *or* pattern *in the context of Pattern Matching is a tuple* $((V_q, E_q), \theta)$*, where* (V_q, E_q) *is a graph and* θ *is a boolean function on the labels and properties of property graphs. The range of* θ *is the set of property graphs. It is w.l.g. a conjunctive normal form* $\theta = \theta_{X_1} \wedge \theta_{X_2} \wedge ... \wedge \theta_{X_n}$*, where* $X_i \subseteq (V_q \cup E_q)$ *and* $X_i \cap X_j = \emptyset \ \forall i, j (1 \leq i, j \leq n)$*. A single clause* X_i *describes all the constraints on some query graph element set* $X_i \subseteq (V_q \cup E_q)$*.*

This definition obviously extends for G to be of any of the data graph types introduced in Section 2.2. All of them equip graph elements with labels and properties. The function θ describes constraints on them, while (V_q, E_q) is a simple graph describing the *structure* of the query pattern. A

[2] `flink.apache.org`, accessed August 2020
[3] `gradoop.com`, accessed August 2020

CNF like θ can be seen as a set of sets of clauses. Hence, throughout the thesis, set operations as well as logical ones are used to denote manipulations of θ.

Definition 10 extends Definition 9 so that time-dependent queries on Temporal Property Graphs are possible.

Definition 10 (Time-dependent Query/Pattern)**.** *A* Time-dependent query *is a query (Definition 9), in which θ may also refer to the properties* tx-from*,* tx-to*,* val-from*,* val-to*.*

A *match* for a query within a data graph G must match the structure of the query as well as its constraints θ. However, there are several different notions of matches, i.e. several types of matching problems. The most important paradigms within this thesis are matches by homomorphism and matches by isomorphism. They are based on graph homomorphisms and isomorphisms, respectively.

Definition 11 (Graph Homomorphism)**.** *Let $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ be graphs. A function $f : V_1 \to V_2$ is a* graph homomorphism *from G_1 to G_2 if f is injective and*

$$(u, v) \in E_1 \Rightarrow (f(u), f(v)) \in E_2 \quad \forall u, v \in V_1$$

Definition 12 (Graph Isomorphism)**.** *Let $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ be graphs. A function $f : V_1 \to V_2$ is a* graph isomorphism *from G_1 to G_2 if f is a bijective graph homomorphism from G_1 to G_2*

Homomorphisms and isomorphisms can now be used to define matches for queries in a property graph.

Definition 13 (Match by Homomorphism (Isomorphism))**.** *A property graph $G = (V, E, T, \tau, K, A, \kappa)$ matches the query $Q = ((V_q, E_q), \theta)$* by homomorphism (isomorphism) *iff the following conditions hold:*

- *There is a graph homomorphism (isomorphism) $f : V_q \to V$, so that*

$$(V, E) := (\{f(v_q) | v_q \in V_q\}, \{(f(u), f(v) | (u, v) \in E_q\})$$

- *$\theta((V, E, T, \tau, K, A, \kappa))$ holds*

These definitions can be easily extended to EPGM and TPGM graphs, as such graphs are property graphs, too.

Figure 2.6 shows an example for a simple query on the data graph in Figure 2.2 and its matches using homomorphism and isomorphism matching paradigms. There are a few things to note about the query in Figure 2.6 and its matches. First, the two matches for the isomorphism paradigm are semantically identical. For *Result1*, Alice's data vertex is mapped the query vertex $v1$ while Bob's data vertex is mapped $v2$. In *Result2*, it is the other way round. Second, it is clear from Definitions 11 and 12 that every isomorphic match is also a homomorphic match. Thus, there are

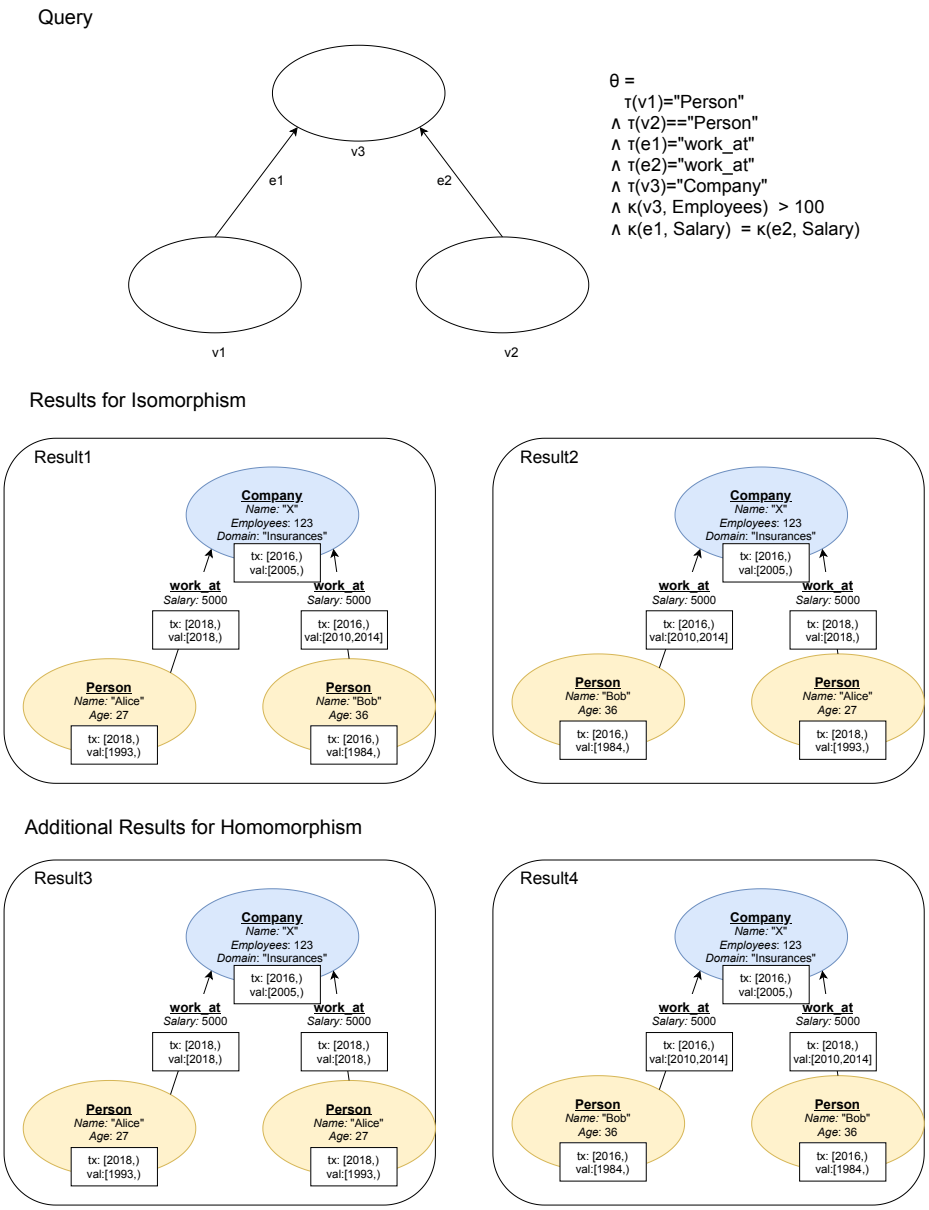

Figure 2.6.: Exemplary query and its results

overall 4 matches using the homomorphism paradigm. The 2 additional results obtained by using the homomorphism paradigm, however, are not particularly useful in this case: in both of them, $v1$ and $v2$ are mapped to the exact same vertex (Alice and Bob respectively), which is permitted for homomorphic matches. A third notable aspect is that the query does not take into account temporal information. Hence, it is never checked, if the two persons actually worked for the company at the same time.

Now, the problem of (Time-dependent) Pattern Matching can be defined.

Definition 14 (Pattern Matching)**.** *Let G be a data graph and Q be a query. The problem of* Pattern Matching *is the task to find every $G' \sqsubseteq G$ where G' matches Q according to some matching paradigm, e.g. isomorphism.*

This definition naturally extends to temporal queries as defined in Definition 10.

Definition 15 (Time-dependent/Temporal Pattern Matching)**.** *Let G be a temporal property graph and Q be a temporal query. The problem of* Time-dependent/Temporal Pattern Matching *is the task to find every $G' \sqsubseteq G$ where G' matches Q according to some matching paradigm, e.g. isomorphism.*

Because matches in the homomorphism and isomorphism paradigm are always subgraphs of the data graph, these paradigms are called *exact* matching paradigms. There are inexact matching paradigms like *Dual Simulation* [42], too, but they are not relevant for this thesis.

Note that the problem of checking for two graphs (V_1, E_1) and (V_2, E_2) whether there is a graph homomorphism $f : V_1 \to V_2$, is NP-complete [11]. It is widely assumed that the same is true for isomorphisms, although this could not be verified yet. However, in practice, modern Pattern Matching algorithms run quite efficient in many cases [37].

3. Related Work

This thesis is related to several different areas of research. Hence, there is plenty of related work that will be shortly discussed in the following.

First, several problems related to Pattern Matching on temporal graphs are introduced in Section 3.1. Then, Section 3.2 lists approaches to general Pattern Matching, followed by temporal Pattern Matching algorithms (Section 3.3). Temporal Pattern Matching implementations in existing graph processing systems and databases are briefly reviewed in Section 3.4. Section 3.5 provides an overview over papers on Gradoop. The related work Section is concluded by a first look at the pattern matching operator already implemented in Gradoop in Section 3.6.

3.1. Problems on Temporal Graphs

Holme and Sarmäki [25] discuss temporal graphs at length and give various real-world examples for 'Temporal Networks' that can be modelled with graphs. They introduce a wide range of different topological characteristics and measurements on temporal graphs and compare several temporal graph models. Other studies on temporal graphs in general are provided by Kostakos [33] and Michail [47]. It should be noted that there is no standard model for temporal graphs.

Many problems and algorithms on static graphs have also been investigated for temporal graphs. Examples include PageRank [59], depth- and breadth-first search [27] and community detection [3]. In the remainder, a few problems closely related to Pattern Matching are introduced. Wu et al. [69] discuss four variations of the 'minimum temporal path' problem. Their model assigns 'starting times' and durations to edges, vertices do not have temporal information. Then, given two vertices u and v, the paths from u to v with the earliest arrival, latest departure, fastest overall time and shortest timespan spent on the edges can be computed. These types of paths as well as reachability in temporal graphs have further been investigated in [70].

Semertzidis and Pitoura [61] study the efficient finding of the most durable matches for a graph query in a temporal graph.

Liu et al. [41] propose an expressive syntax to query temporal knowledge graphs. This syntax allows to search for keywords in the graph data and to impose constraints on temporal relations between graph elements. To process such queries, generalizations of Dijkstra's algorithm are used.

Kovanen et al. [34] introduce the notion of 'motifs' in temporal graphs. For static graphs, a motif is defined as a connected subgraph induced by a set of edges. Kovanen et al. [34] adapt this definition to temporal graphs, including temporal aspects of connectivity. Moreover, they present an algorithm to identify such temporal motifs. Algorithms to count temporal motifs are given by Paranjape et al. [49].

3.2. Pattern Matching Algorithms

Conte et al. [11] provide an overview over various graph pattern matching approaches from 1974 to 2004. They distinguish between exact and inexact matching algorithms. Exact matching preserves adjacency. Inexact matching is motivated by exact matching being a NP-complete problem. Relaxing this problem allows to obtain approximate solutions efficiently. Recent examples for such an approach are Fan et al. [14] and Ma et al. [43]. Matches by homomorphism and isomorphism as defined in Definition 13 are exact. Hence, inexact matching paradigms will not be treated in the following. Furthermore, there are algorithms that only check whether there is a match for the query in the data graph. As the pattern matching operator in this thesis aims at finding all matches, these algorithms are not relevant here, either. The most well-known pattern matching algorithm was introduced by Ullmann in 1976. Ullmann's algorithm is a tree-search algorithm using backtracking and pruning heuristics. Several more recent subgraph isomorphism algorithms can be seen as improving Ullmann. Examples are VF2 [12], the GraphQL pattern matching algorithm by He and Singh [24], QuickSI [63], GADDI [73] and SPath [74]. Lee et al. [37] compare them, integrating them into a common framework. However, the data sets used to compare the performance of the algorithms are all quite small. While in Ullmann's algorithm, the vertex set is traversed in random order when searching for matches, all these algorithms aim at finding a more efficient order. Furthermore, they employ more refined pruning procedures and/or indices. TurboISO [22] rewrites queries to so-called NEC (neighborhood equivalence class) trees in order to explore the search space more efficiently. Ren and Wang [55] propose a method to speed up all these backtracking algorithms, which are similar to Ullmann's. To achieve this, they leverage certain relations between data graph vertices, prohibiting the processing of too much duplicates. Bi et al. [4] further refine the aforementioned algorithms by taking Cartesian products of partial matches into account and employing a data structure for paths. Another Ullmann-based approach to subgraph isomorphism is proposed by Saltz et al. [60], who use the inexact matching paradigm DualSim [42] to prune the search tree. VF3 [6] is a recent improvement of VF2, especially suitable for large and dense graphs.

Different from these algorithms, there are approaches relying mostly on graph indices. Yan et al. [71] propose the indexing method gIndex and show how it can be used for pattern matching. FG-Index by Cheng et al. [8] indexes frequent subgraphs. If a query matches such a sub graph, the results can be immediately returned. Otherwise, additional checks need to be conducted.

Only few subgraph isomorphism algorithms, however, are explicitly designed for distributed execution. Fard et al. [15] introduce several distributed pattern matching algorithms using the vertex-centric programming paradigm made prominent by Pregel [44]. However, all of them only handle inexact matching paradigms. D-ISI by Wickramaarachchi et al. [68] is one of the very few distributed subgraph isomorphism approaches. It allows Pattern Matching on dynamic graphs which should have a small diameter for efficient processing.

None of the algorithms above is limited to a certain Graph Data Model, they could, in theory, be adapted to all graph data models described in Section 2.2. However, also none of them focuses on temporal graphs and specific temporal queries.

3.3. Temporal Pattern Matching Algorithms

There are not as many algorithms concerned with temporal pattern matching. A problem in comparing them is that there is no widely accepted definition of temporal graphs. Thus, the algorithms introduced in the following sometimes differ w.r.t. their graph models.

Redmond and Cunningham [54] propose an approach based on VF2 that finds patterns which must be time-respecting. A path being 'time-respecting' means that the temporal values on the edges of the path do not decrease, vertices do not have time stamps in this model.

Similar to Redmond and Cunningham [54], Semertzidis and Pitoura [62] assume only edges to have time stamps in their model. They aim to match 'interaction patterns' in which the temporal order of edges in a match must be preserved. Furthermore, interaction patterns are restricted to occur within a period of certain length. Their algorithm incrementally matches edges using an index.

In the model by Franzke et al. [17], only edges are associated with an interval, too. Two approaches are used for temporal query processing. Specific temporal pruning methods are applied to improve Ullmann's basic algorithm [65]. Moreover, an index structure is employed. Queries are limited to arbitrary graphs where patterns on the edge intervals are specified.

Chen et al. [7] investigate temporal social networks, in which every vertex and edge is associated with an interval. However, they only provide algorithms for matching three special classes of queries, e.g. 'Group of Users with Relationship Duration (GURD)' queries.

Different from all these approaches, Li et al. [39] present an isomorphism matching algorithm for streaming graphs. In such graphs, edges arrive continuously and their arrival time is considered their time stamp. Vertices do not have time stamps in this model. Queries are graphs where the temporal order of edges, i.e. the order in which they arrived, is specified. Their algorithm employs a data structure similar to Tries. Furthermore, it is capable of processing incoming edges in parallel. Song et al. [64] also provide a method for pattern matching on streaming graphs. Their algorithm preserves the temporal order of edges, too. However, the matching paradigm is not isomorphism, but inexact graph simulation.

3.4. Temporal Pattern Matching in Graph Processing Systems and Graph Databases

In this Section, the differentiation between (distributed) Graph Processing Systems and Graph Databases (cf. Junghanns et al. [30]) is adopted. In Graph Databases, a pattern matching operator is typically implemented, as the user should be able to query the database. Further graph mining and analysis methods are usually not supported. However, graph databases provide typical database functions, e.g. updating or deleting data. Graph Processing Systems, on the other hand, often provide a wide range of graph analysis methods, e.g. PageRank, Frequent Subgraph Mining etc. Other than graph databases, which do not scale well, Graph Processing Systems are usually able to handle large amounts of data. A disadvantage is that they do not support persistent storage and

semantically expressive graph data models like those introduced in Section 2.2. See Junghanns et al. [30] for a thorough discussion of these two types of systems. In what follows, temporal pattern matching in various graph processing systems (Section 3.4.1) and graph databases (Section 3.4.2) is briefly reviewed.

3.4.1. Temporal Pattern Matching in Graph Processing Systems

There are many Graph Processing Systems available today, but only a minority of them are shipping with explicit support for temporal graphs. These systems will now be briefly discussed, without heir architectures being introduced in detail.

ImmortalGraph [46], formerly known as Chronos [21] is an early distributed temporal graph processing system often referenced in related work. ImmortalGraph's source code is not available.

Similar to ImmortalGraph, Kineograph [9] is based on snapshots, so that vertices and edges are not explicitly assigned intervals.

Chronograph [5] is based on the PGM (Def. 5). Its temporal semantics include both intervals and points in time. Its main focus is the traversal of temporal graphs. For the traversals, constraints on the temporal sequence as introduced by Allen [2] can be stated. These relations will also play an important role in this thesis, see Section 4.3.2.

Interval semantics are also to be found in Tink's [40] data model. However, only edges have intervals here. Tink is based on Gelly[4], which is, in turn an abstraction of Apache Flink for graphs.

TGraph [26] defines a different kind of temporal graph model, in which properties of vertices and edges are time-dependent. TGraph extends Neo4J.

In Graphite [18], intervals also play a key role.

However, none of these systems supports pattern matching as defined in Section 2.4.

3.4.2. Temporal Pattern Matching in Graph Databases

Processing queries is a core feature of (Graph) Databases. In the following, a few graph databases supporting temporal queries are characterized. The focus is, again, not on architecture or implementation details, but on temporal query processing features. Graph databases can be roughly classified by the data model they support [30]: the most widespread data models are the PGM and RDF [10].

RDF's data model does not include any mandatory temporal information. Nevertheless, RDF data may contain temporal information in XML-Schema datatypes such as `date`. The query language for RDF is SPARQL [23], which is standardized, too (currently in the version 1.1). SPARQL provides functions on `xsd:datetime`s to extract the specified year, month, hour etc. Timezones can also be handled. Furthermore, values of type `xsd:datetime` can be compared using the comparators `=`, `!=`,

[4] `https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/gelly/`, accessed September 2020

`>`, `>=`, `<` and `<=`. Examples for graph databases based on RDF are AllegroGraph [1] and TripleBit [72].

As pointed out in Section 2.2, the PGM does not prescribe temporal information either.

Neo4J [48] is a very well-known graph database using the PGM. Its query language is Cypher [16], which is also the basis of GDL, the query language of Gradoop's existing Pattern Matching operator (see Section 3.6). Cypher supports several date/time data types, e.g. `Date` and `DateTime`. There is a rich variety of options to specify such times and dates. The number of hours, minutes etc. in a time value can be acessed. Moreover, Cypher includes a `Duration` datatype. Durations can be created similar to date and time types. Arithmetic operations on them are possible. Dates can be compared with the usual comparators like `<`. Furthermore, new dates can be created by subtracting or adding a duration to an existing date. The duration between two dates can be computed and used in query predicates, too.

PGQL [66], the query language of Oracle Big Data Spatial and Graph[5], extends SQL in order to query property graphs. Thus, it supports SQL's `DATE` type and comparisons involving it.

OrientDB[6] includes a property graph model. It is also able to handle the `DATE` type of SQL.

In Sparksee [45], a `Timestamp` type is provided, too.

To conclude, many graph databases support temporal data. However, none of them explicitly includes temporal information in its data model.

3.5. Gradoop

An overview over the architecture of Gradoop, EPGM and GrALa is provided by Junghanns et al. [32]. TPGM is introduced by Rost et al. [57].

Several operators and their implementations are discussed in separate papers. Grouping is treated by Junghanns et al. [31], Frequent Subgraph Mining by Petermann et al. [53]. Junghanns et al. [28] present the EPGM Pattern Matching implementation that the work of this thesis is built on. Kricke et al. [35] introduce structural transformations of graphs, e.g. inverting edges or converting properties into vertices. Graph Sampling is described by Gomez et al. [20].

Furthermore, some applications and examples of Gradoop workflows are given. Petermann and Junghanns [51] as well as Petermann et al. [52] demonstrate how Gradoop can be used for Business Intelligence applications. Further GrALa workflows are demonstrated by Junghanns et al. [29]. A temporal analysis use case is shown by Rost et al.[56].

Rostami et al.[58] built a visual interface for Gradoop workflows.

[5] `https://www.oracle.com/database/technologies/bigdata-spatialandgraph.html`, accessed September 2020
[6] `http://orientdb.org/docs/3.0.x/`, accessed August 2020

3.6. The Pattern Matching Operator in Gradoop

Gradoop implements a pattern matching approach that is inspired by query processing in relational DBMS. Its implementation is described in detail in [28] and Section 5 of this thesis.

Pattern Matching is among the operators defined by EPGM as well as by TPGM. However, the implementation is not yet capable of processing time-dependent queries. The main part of this thesis is devoted to the implementation of time-dependent pattern matching in Gradoop. To achieve this, the already implemented EPGM pattern matching program is extended.

Before this extension is described at length in the following chapters, the existing operator is introduced from a user's perspective.

Like all operators in Gradoop, Pattern Matching is accessible via the Java API for GrALa. Listing 3.1 displays a GrALa program that executes the query from Figure 2.6 on the data graph `graph` and prints the resulting graph collection. Alternatively, the matches could have been written to a data sink.

```
1  LogicalGraph graph = // ...
2
3  String query = "MATCH " +
4                 "(v1:Person)-[e1:work_at]->(v3:Company) " +
5                 "(v2:Person)-[e2:work_at]->(v3) " +
6                 "WHERE " +
7                 "v3.Employees > 100 " + "AND " +
8                 "e1.Salary = e2.Salary ";
9
10 GraphCollection results = graph.query(query);
11
12 results.print();
```

Listing 3.1: Exemplary GrALa Pattern Matching

The query (lines 2-8 of Listing 3.1) is described in the Graph Description Language (GDL). GDL is an open source[7] query language that was created for Pattern Matching in Gradoop [28]. It was designed to resemble Cypher [16], the query language of Neo4J.

GDL queries always start with the keyword `MATCH`, followed by a graph pattern. Vertices are denoted by round brackets. The round brackets may contain the variable name for the vertex. The label of the vertex can be specified, too. Both variable name and label are optional, e.g. `()`, `(v1)`, `(:Person)` and `(v1:Person)` are all valid query vertices. Edges are declared as arrows `-->`. They might contain a squared bracket in which an optional variable name and a label can be declared in the same way as in vertices. Additionally, paths of variable length can be specified via *path expressions*. E.g., `(p1)-[e:knows*1..3]->(p2)` matches any path between two vertices consisting of one to three *knows* edges.

[7] `github.com/s1ck/gdl`, accessed August 2020

The query constraints on properties are optionally specified in a SQL-like `WHERE`-clause. Here, properties of graph elements (selected in the form `variable.property`) can be compared with each other or to suitable literals. Such comparisons can be combined to a complex boolean expression via the keywords `AND`, `OR`, `XOR` and `NOT`.

In addition, a construction pattern can be provided to transform the matching results. An example is shown in Listing 3.2. It modifies line 10 of Listing 3.1 by adding a construction pattern which leads to only the two *Person* vertices being returned for each match. Moreover, they are now connected via a new type of edge, *sameCompany*, meaning that these two persons work(ed) at the same company some time.

```
1  LogicalGraph graph = // ...
2
3  String query = //...
4
5  String constructionPattern = "(p1)-[e:sameCompany]->(p2)";
6
7  GraphCollection results = graph.query(query, constructionPattern);
```

Listing 3.2: GrALa Pattern Matching with construction pattern

The task of extending the Pattern Matching operator to support temporal queries is two-fold: First, GDL must be extended so that queries can refer to transaction and valid intervals. Second, the Gradoop operator must be able to process these new types of queries. The temporal extensions of GDL are presented in Section 4. Section 5 describes the existing Pattern Matching implementation in Gradoop and its adaptation to temporal queries.

4. GDL Extension

The structure of Temporal Property Graphs does not differ from that of Extended Property Graphs. Hence, the `MATCH` clause in GDL queries does not need to be extended. An extension of GDL in order to enable time-related queries can be confined to the `WHERE` clause.

Temporal Queries must be able to compare the four timestamps *tx-from*, *tx-to*, *val-from*, *val-to* among each other and with timestamp literals. Thus, there must be selectors for the timestamps as well as a literal type for timestamps. Section 4.1 lists these extensions.

Another desideratum in temporal queries is to check the intervals (*transaction*, *valid*) of different graph elements for certain relations between them. This extension is presented in the Sections 4.2 and 4.3.

A further important aspect of intervals is their duration. Comparisons between interval durations are supported by the GDL extension, too, as presented in Section 4.4.

Section 4.5 lists some exemplary queries that illustrate the expressiveness of the proposed syntax extensions.

The introduction of the extensions is succeeded by a few remarks on GDL's implementation (Section 4.6).

Note that support for temporal constraints on path expressions is not within the scope of this thesis. Section 4.7 discusses the reasons for this.

4.1. Time Stamps

In what follows, all options to create time stamps, i.e. references to exactly one point in time, are explained. For a compact overview, refer to Table 4.1 in Section 4.1.5.

4.1.1. From and To Selectors

Intervals like the transaction and valid intervals in TPGM are described by two timestamps, their start/from timestamp and their end/to timestamp. It is thus essential for temporal queries to be able to refer to the four timestamps *tx-from*, *tx-to*, *val-from*, *val-to* that are associated with every vertex and ege in TPGM. In the GDL extension, the four time stamps can be accessed in the same syntax as any other element property, e.g. `a.tx_from`.

Different from the notation in this text, *tx-from* (and analogously the other time stamps) is denoted `tx_from` in the GDL extension in order not to use the hyphen. Even though GDL does not support arithmetic expressions at the moment, the hyphen might be used as a minus operator in the future without causing confusion with the time selectors.

These selectors can now be used to compare transaction and valid intervals of different graph elements. E.g., a constraint `a.tx_from < b.tx_from` states that the transaction interval of some

element `a` must start before the transaction interval of some `b`. Put differently, `a` must have been added to the database before `b`.

4.1.2. Time Stamp Literals

In many cases, it is necessary to compare a from or to selector to a certain date or time. For example, one might only be interested in elements that were added to the database in 2020 or later. This constraint can be expressed as `a.tx_from > Timestamp(2020-01-01)`. The GDL extension permits the creation of time stamp literals like `Timestamp(2020-01-01)` or `Timestamp(2020-01-01T12:01:55)`. Moreover, the keyphrase `Timestamp(now)` creates a time stamp referencing the current system time. Table 4.1 lists the three options in a more systematic way.

Time Stamps with respect to different time zones have not been addressed in this thesis. All time stamps created in GDL have GMT as their time zone.

4.1.3. Global From and To Selectors

TPGM specifies transaction and valid intervals for every single graph element. Based on these 'local' intervals, 'global' transaction and valid intervals, associated with the whole graph can be defined.

Definition 16 (Global transaction interval)**.** *Let G be a temporal property graph with vertices V and edges E. Then,*

$$\text{G.tx-from} := max_{x \in (U \cup V)} \text{x.tx-from}$$
$$\text{G.tx-to} := min_{x \in (U \cup V)} \text{ x.tx-to}$$

The global transaction interval *of G, denoted* G.tx *is defined as*

$$[\text{G.tx-from}, \text{ G.tx-to}]$$

, if and only if G.tx-from $\leq$ G.tx-to*.*

Because of the latter condition, every comparison c involving the global transaction interval is transformed to $c \wedge (max_{x \in U \cup V} x.tx\text{-}from \leq min_{x \in U \cup V}\ x.tx\text{-}to)$ using MIN and MAX expressions as introduced in Section 4.1.4.

Analogous definitions and remarks hold for G's *global valid interval* (denoted $G.val$).

4.1.4. MIN and MAX expressions

In some cases it is useful to determine the earliest or latest time point among a set of selectors and/or time stamps. For example, global transaction and valid times for subgraphs of the query graph can be declared this way. If a query graph contains elements a, e and b, but only the global valid interval for the subgraph containing only a and b is of importance, this valid interval is defined by [*max{a.val-from, b.val-from}, min{a.val-to, b.val-to}*]. MIN and MAX timestamps can be created using the syntax `MIN(t1,...,tn)` and `MAX(t1,...,tn)`, where `ti` ($1 \leq i \leq n$) is some time stamp other than a MIN/MAX expression. Hence, MIN/MAX expressions can not be nested but take only time stamps and from/to selectors as arguments.

4.1.5. Overview over Timestamp Creation

Table 4.1 lists all possibilities to create a time stamp in the GDL extension.

Type	Syntax	Remarks
valid-from (variable)	`variable.val_from`	
valid-to (variable)	`variable.val_to`	
tx-from (variable)	`variable.tx_from`	
tx-to (variable)	`variable.tx_to`	
valid-from (global)	`val_from`	implicitly adds *val-from* $\leq$ *val-to*
valid-to (global)	`val_to`	implicitly adds *val-from* $\leq$ *val-to*
tx-from (global)	`tx_from`	implicitly adds *tx-from* $\leq$ *tx-to*
tx-to (global)	`tx_to`	implicitly adds *tx-from* $\leq$ *tx-to*
Datetime Literal	`Timestamp(YYYY-MM-DDThh:mm:ss)`	
Date Literal	`Timestamp(YYYY-MM-DD)`	sets time to 00:00:00
Minimum	`MIN(t1,...,tn)`	$n \geq 2$, no nesting of MIN/MAX
Maximum	`MAX(t1,...,tn)`	$n \geq 2$, no nesting of MIN /MAX

Table 4.1.: Options to create time stamps in GDL queries

4.2. Intervals

Apart from time stamps, intervals can be referenced in queries. This Section introduces all options to create intervals in a `WHERE` clause. An overview over them is given in Table 4.2 in Section 4.2.4. Section 4.3 then explains how intervals can be compared to each other and to time stamps.

4.2.1. Transaction and Valid Intervals

For every query graph element, its transaction and valid interval can be accessed in a property selector syntax, e.g. `a.val` denotes the valid interval of some graph element.

In addition, queries can refer to global intervals. For example, `tx` denotes the global valid interval.

4.2.2. Custom Intervals

To enable flexible creation of custom intervals, a keyword `Interval` is introduced to GDL. The expression `Interval(x,y)` creates an interval $[x, y]$. Here, x and y are timestamps as defined in Section 4.1. Heterogeneous combinations of time stamps like `Interval(a.tx_from, Timestamp(2020-01-01))` are possible. For an expression `Interval(x,y)`, a constraint $x \leq y$ is implicitly added to the respective query clause to ensure that the created interval is actually defined.

4.2.3. Merging and Joining Intervals

A third option to create intervals in a query is to merge or join two intervals. These two operations are only defined, when the involved intervals overlap.

Definition 17 (Overlapping Intervals)**.** *Let* i_1 =[a,b] *and* i_2 =[c,d] *be intervals. They are said to* overlap*, if and only if* max{a,c} $<$ min{c,d}.

The merge of overlapping intervals can be thought of as the interval defined by their intersection, while their join corresponds to the interval defined by their union.

Definition 18 (Merge and Join of Intervals)**.** *Let* $i_1 = [i_1\_from, i_1\_to]$, $i_2 = [i_2\_from, i_2\_to]$,...,$i_n = [i_n\_from, i_n\_to]$ *be overlapping intervals.*
Their merge *is defined by the interval*

$$[\max\{i_1\_from, i_2\_from,...,i_n\_from\}, \min\{i_1\_to, i_2\_to,...,i_n\_to\}]$$

Their join *is defined by the interval*

$$[\min\{i_1\_from, i_2\_from,...,i_n\_from\}, \max\{i_1\_to, i_2\_to,...,i_n\_to\}]$$

Figure 4.1 illustrates this definition.

In GDL, merge and join intervals are created by expressions `i1.merge(i2)` and `i1.join(i2)`. Here, `i1` and `i2` are primitive intervals, i.e. transaction/valid intervals (Section 4.2.1) or custom intervals (Section 4.2.2). In other words, merge and join intervals can not be nested. Both operators are obviously commutative.

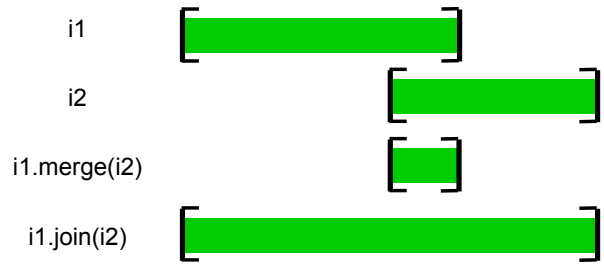

Figure 4.1.: Illustration of interval merge/join

Type	Syntax	Remarks
tx interval (variable)	`var.tx`	
valid interval (variable)	`var.val`	
tx interval (global)	`tx`	implicitly adds *tx-from* $\leq$ *tx-to*
valid interval (global)	`val`	implicitly adds *val-from* $\leq$ *val-to*
custom interval	`Interval(t1, t2)`	*t1, t2* time stamps, creates [*t1, t2*], implicitly adds $t1 \leq t2$
merge interval	`i1.merge(i2)`	implicitly adds constraints enforcing overlap of *i1* and *i2*
join interval	`i1.join(i2)`	implicitly adds constraints enforcing overlap of *i1* and *i2*

Table 4.2.: Options to create intervals in GDL queries

Although merge and join are n-ary operations, GDL only supports merges and joins of two intervals in order to keep the grammar simple. N-ary merges and joins can be easily expressed by creating a custom interval (see 4.2.2) with MIN and MAX timestamps (see 4.1.4). Actually, the *merge* and *join* syntax presented here is only syntactic sugar for such a construction.

4.2.4. Overview over Interval Creation

In Table 4.2, all options for creating intervals are listed.

4.3. Constraints on Intervals and Time Stamps

Once time stamps and intervals are created, constraints on them can be formulated. Section 4.3.1 presents all possibilities to compare time stamps to other time stamps or intervals. After that, relations between intervals and their syntax in GDL are discussed in Section 4.3.2.

4.3.1. Comparisons involving Time Stamps

As already mentioned in Section 4.1, time stamps can be compared among each other just like any other property selectors and literals. For example, comparisons like
`a.tx_from > Timestamp(2020-01-01)` or

`e.tx_from <= MIN(a.val_from, b.val_from)` are possible. Available comparators are `<`, `<=`, `=`, `!=`, `>=` and `>`.

Moreover, there are aliases for the comparators `<` and `>`: `t1.before(t2)` is equivalent to `t1<t2`, while `t1.after(t2)` equals `t1>t2` for time stamps `t1`, `t2`.

Time stamps can not only be compared among each other but also to intervals. In the GDL extension, three possible relations (and their negations via `NOT`) between a time stamp t and an interval $i = [i_{from}, i_{to}]$ can be formulated:

- t can denote a point in time before i starts, i.e. $t < i_{from}$. This relation is expressed `t.precedes(i)` in GDL
- t might refer to a point in time that is contained in i, i.e. $i_{from} \leq t \leq i_{to}$. This relation is stated `i.contains(t)` in GDL
- t can represent a point in time after i ends, i.e. $t > i_{to}$. The GDL syntax to express this relation is `t.after(i)`

Obviously, these keywords are only syntactic sugar. The relations can be formulated using only comparisons of simple time stamps.

Additionally, predicates `fromTo` and `between` are provided, as defined in [57]. These predicates are inspired by the keywords `FROM...TO` and `BETWEEN` available in SQL:2011 [36]. They relate an interval $i = [i_{from}, i_{to}]$ to two time stamps *t1* and *t2* that can be viewed to describe intervals:

- `i.fromTo(t1,t2)` := $i_{from} < t2 \wedge i_{to} > t1$. This means that i must overlap with the interval $[t1, t2)$
- `i.between(t1,t2)` := $i_{from} \leq t_2 \wedge i_{to} > t_1$. Here, an overlap of i and the interval $[t1, t2]$ is postulated

As before, the predicates `fromTo` and `between` are just aliases for the equivalent expressions over time stamps.

Another relation between an interval and a timestamp is the *asOf* relation. It expresses the constraint that an element must have been present/valid in the database at a certain point in time. E.g., `a.tx.asOf(Timestamp(2020-01-01))` states that some element a was in the database as of 2020-01-01 at midnight. The *asOf* relation is syntactic sugar, too: `a.asOf(t)` is equivalent to *a.tx-from* $\leq t \wedge$ *a.tx-to* $\geq t$. An `AS OF` keyword is also defined in SQL:2011.

Table 4.3 lists all the comparisons involving time stamps that were presented here.

Lukas Christ
3756213

Syntax	Semantics
`t1 < / <= / = / != / >= / > t2`	$t1 < / \leq / = / \neq \ / \geq / > \ t2$
`t1.before(t2)`	$t1 < t2$
`t1.after(t2)`	$t1 > t2$
`t1.precedes(i)`	$t1 < i_{from}$
`t1.after(i)`	$t1 > i_{to}$
`i.contains(t1)`	$i_{from} \leq t \leq i_{to}$
`i.fromTo(t1, t2)`	$i_{from} < t2 \ \wedge \ i_{to} > t1$
`i.between(t1, t2)`	$i_{from} \leq t2 \ \wedge \ i_{to} > t1$

Table 4.3.: Comparisons involving time stamps in GDL queries

4.3.2. Interval Relations

All intervals described in Section 4.2 can not only be compared to a timestamp but also among each other. However, the comparators ($<$, $\leq$,...) are not suited to compare two intervals. Instead, two intervals can have certain relations.

Allen [2] describes 13 binary interval relations. They are listed in Table 4.4 for intervals $[a, b]$ and $[c, d]$. None of Allen's relations, except the equality relation, are symmetric. For every relation, its

Relation	Semantics	Inverse
$[a, b]$ before$_A$ $[c, d]$	$b < c$	after$_A$
$[a, b]$ after$_A$ $[c, d]$	$a > d$	before$_A$
$[a, b]$ during$_A$ $[c, d]$	$a > c \wedge b < d$	contains$_A$
$[a, b]$ contains$_A$ $[c, d]$	$a < c \wedge b > d$	during$_A$
$[a, b]$ overlaps$_A$ $[c, d]$	$a < c < b < d$	overlapped-by$_A$
$[a, b]$ overlapped-by$_A$ $[c, d]$	$c < a < d < b$	overlaps$_A$
$[a, b]$ meets$_A$ $[c, d]$	$b = c$	met-by$_A$
$[a, b]$ met-by$_A$ $[c, d]$	$a = d$	meets$_A$
$[a, b]$ starts$_A$ $[c, d]$	$a = c \wedge b < d$	started-by$_A$
$[a, b]$ started-by$_A$ $[c, d]$	$a = c \wedge b > d$	starts$_A$
$[a, b]$ finishes$_A$ $[c, d]$	$a > c \wedge b = d$	finished-by$_A$
$[a, b]$ finished-by$_A$ $[c, d]$	$a < c \wedge b = d$	finishes$_A$
$[a, b]$ equals$_A$ $[c, d]$	$a = c \wedge b = d$	equals$_A$

Table 4.4.: Allen's [2] interval relations

inverse is also defined as a relation.

SQL:2011 defines interval relations ('period predicates'), too [36]. These relations are inspired by those proposed by Allen. However, there are only 7 interval relations in SQL:2011. Table 4.5 lists them for intervals $[a, b]$ and $[c, d]$ and also shows their equivalent expressions using Allen's relations (cf. [36]). The difference between Allen's overlaps$_A$ relation and the `OVERLAPS` relation in SQL is especially notable. The latter is equal to the definition of overlapping intervals given in Definition 17.

For the GDL extension, interval relations equivalent to those in SQL are provided. They are listed in Table 4.6.

SQL Relation	Semantics	Allen Equivalent
$[a, b]$ `OVERLAPS` $[c, d]$	$\max\{a, c\} < \min\{b, d\}$	$[a, b]$ overlaps_A $[c, d]$ $\vee$ $[a, b]$ overlapped_by_A $[c, d]$ $\vee$ $[a, b]$ during_A $[c, d]$ $\vee$ $[a, b]$ contains_A $[c, d]$ $\vee$ $[a, b]$ starts_A $[c, d]$ $\vee$ $[a, b]$ started_by_A $[c, d]$ $\vee$ $[a, b]$ finishes_A $[c, d]$ $\vee$ $[a, b]$ finished_by_A $[c, d]$ $\vee$ $[a, b]$ equals_A $[c, d]$ $\vee$
$[a, b]$ `CONTAINS` $[c, d]$	$a \leq c \wedge b \geq d$	$[a, b]$ contains_A $[c, d]$ $\vee$ $[a, b]$ starts_by_A $[c, d]$ $\vee$ $[a, b]$ finishes_A $[c, d]$ $\vee$ $[a, b]$ equal_A $[c, d]$ $\vee$
$[a, b]$ `PRECEDES` $[c, d]$	$b \leq c$	$[a, b]$ before_A $[c, d]$ $\vee$ $[a, b]$ meets_A $[c, d]$ $\vee$
$[a, b]$ `SUCCEEDS` $[c, d]$	$b \leq c$	$[a, b]$ after_A $[c, d]$ $\vee$ $[a, b]$ met_by_A $[c, d]$ $\vee$
$[a, b]$ `EQUALS` $[c, d]$	$a = b \wedge c = d$	$[a, b]$ equals_A $[c, d]$
$[a, b]$ `IMMEDIATELY_PRECEDES` $[c, d]$	$b = c$	$[a, b]$ meets_A $[c, d]$
$[a, b]$ `IMMEDIATELY_SUCCEEDS` $[c, d]$	$a = d$	$[a, b]$ met_by_A $[c, d]$

Table 4.5.: Interval relations in SQL:2011 and equivalent Allen [2] formulas

SQL Relation	GDL Predicate
$i1$ `OVERLAPS` $i2$	`i1.overlaps(i2)`
$i1$ `CONTAINS` $i2$	`i1.contains(i2)`
$i1$ `PRECEDES` $i2$	`i1.precedes(i2)`
$i1$ `SUCCEEDS` $i2$	`i1.succeeds(i2)`
$i1$ `EQUALS` $i2$	`i1.equals(i2)`
$i1$ `IMMEDIATELY_PRECEDES` $i2$	`i1.immediatelyPrecedes(i2)`
$i1$ `IMMEDIATELY_SUCCEEDS` $i2$	`i1.immediatelySucceeds(i2)`

Table 4.6.: Interval relations in GDL

4.4. Durations and Duration Constraints

In many cases it may be necessary to impose constraints on the length of intervals, i.e. their durations. For example, in analysing a social network like in Figure 2.2, one might only be interested in long-term employees that have been with the company for at least five years. A condition `e.val.lengthAtLeast(Days(1825))` may describe this constraint for a *work_at* edge *e*. Such conditions are possible in GDL.

Two types of durations can be used to formulate them:

- within the context of a duration constraint, an interval term (cf. Section 4.2) represents the duration of the respective interval.

- constant durations can be created with the self-explanatory keywords `Millis`, `Seconds`, `Minutes`, `Hours` and `Days`. Similar to Java constructors, they take a number as an argument, e.g., `Minutes(5)` creates a constant duration representing 5 minutes.

Table 4.7 summarizes these options.

Duration	Syntax
duration of interval i	(syntax of i as described in Section 4.2)
n milliseconds	`Millis(n)`
n minutes	`Minutes(n)`
n hours	`Hours(n)`
n days	`Days(n)`

Table 4.7.: Options to create durations in GDL queries

Two durations can be compared using the keywords `longerThan`, `lengthAtLeast`, `shorterThan` and `lengthAtMost`. For example, `a.tx.longerThan(b.tx)` states that some element a must have been longer in the database than another element b. Table 4.8 lists these expressions more formally.

Syntax	Semantics
`i1.longerThan(i2)`	$\text{length}(i1) > \text{length}(i2)$
`i1.lengthAtLeast(i2)`	$\text{length}(i1) \geq \text{length}(i2)$
`i1.shorterThan(i2)`	$\text{length}(i1) < \text{length}(i2)$
`i1.lengthAtMost(i2)`	$\text{length}(i1) \leq \text{length}(i2)$

Table 4.8.: Options to compare durations in GDL queries

Note that it is not possible to compare durations using the comparators `<`, `<=`, `!=`, `=`, `>` and `>=`. The reason why there are no keywords `Years` and `Months` is that years and months may differ in their length, whereas milliseconds, seconds, minutes and days do not vary w.r.t their duration.

4.5. Exemplary Queries

To conclude the introduction of temporal predicates in GDL queries, this Section contains a few examples. All of them are intended for the isomorphism paradigm.

Using the `overlaps` predicate, it is now possible to find the pattern of two persons that worked at the same company at the same time in the graph of Figure 2.2 (Listing 4.1).

```
1 MATCH     (p1:Person)-[w1:work_at]->(c:Company)
2           (c)<-[w2:work_at]-(p2:Person)
3 WHERE     w1.val.overlaps(w2.val)
```

Listing 4.1: Example Query 1

In order to find all `works_at` relations that are actually valid now, the `asOf` predicate can be employed (Listing 4.2).

```
MATCH  (p:Person) -[w:works_at]-> (c:Company)
WHERE  w.val.asOf(Timestamp(Now))
```

Listing 4.2: Example Query 2

The next query, Listing 4.3, detects all persons that studied at least two years at the University of Leipzig in the 1990s and earn at least 100k Euros per year today.

```
MATCH  (p:Person) -[s:study_at]-> (u:University)
       (p) -[w:work_at]-> (c:Company)

WHERE  u.Name="University of Leipzig" AND
       s.val.between(Timestamp(1990-01-01),Timestamp(1999-12-31)) AND
       s.val.lengthAtLeast(Years(2)) AND
       w.Salary > 100000 AND
       w.val.asOf(Timestamp(Now))
```

Listing 4.3: Example Query 3

For the last example, suppose the company 'X' wants to find mentors for its new employees. A new employee is one that started working for 'X' in 2020. Potential Mentors are at most 15 years (5475 days) older than their mentees and have been with the company for at least 5 years (1825 days). Listing 4.4 shows a possible query to find pairs of possible mentors and mentees.

```
MATCH  (mentor:Person) -[w1:work_at]-> (c:Company)
       (mentee:Person) -[w2:work_at]-> (c)

WHERE  c.Name="X" AND
       w2.val_from >= Timestamp(2020-01-01) AND
       Interval(mentor.val_from, mentee.val_from).lengthAtMost(Days(5475)) AND
       w1.val.lengthAtLeast(Days(1825)) AND
       val.asOf(Timestamp(Now))
```

Listing 4.4: Example Query 4

These examples demonstrate the expressivity of the implemented GDL syntax extensions. The next Section briefly sketches the most important aspects of their implementation in GDL.

4.6. Implementation of GDL

GDL is based on an ANTLRv4 ([50]) grammar. To give an example, the rules describing merges and joins for intervals are shown in Listing 4.5.

```
complexInterval
    : complexIntervalArgument '.merge(' complexIntervalArgument ')'
    | complexIntervalArgument '.join(' complexIntervalArgument ')'
    ;

```

```
6  complexIntervalArgument
7      : intervalSelector
8      | intervalFromStamps
9      ;
```

Listing 4.5: Sample from GDL ANTLR grammar

Here, `complexInterval`, `complexIntervalArgument`, `intervalSelector` and `intervalFromStamps` are non-terminal variables. Strings in quotes (here: `.merge(`, `.join(` and twice `)`) are constants. The non-terminal `intervalSelector` describes interval expressions referring to transaction and valid intervals, while `intervalFromStamps` is used to define custom intervals (Section 4.2.2).

The query parsing is implemented in Java. Therefore, the query is wrapped into Java objects. Classes like `Graph.java`, `Edge.java` and `Vertex.java` wrap the pure structure of the specified graph, i.e. just the vertices and edges without any information about properties and labels. All constraints are given in a boolean formula. Here, comparisons between two instances of `Comparable.java` are the atomic constraint types. An interface `ComparableExpression.java` is extended by all wrappers for comparable data types. These types are, without the temporal extensions, `PropertySelector.java`, `Literal.java` and `ElementSelector.java`. The latter selects a vertex or edge.

The extensions of GDL described above extend the grammar and introduce new types of `ComparableExpression.java`. A new abstract implementation of it, `TimePoint.java` is created as a common base class for all temporal data types. Changing the wrappers for the graph structure or the boolean formula is not necessary.

The `TimePoint` type `TimeSelector` is used to wrap all from and to selectors (Section 4.1.1). Time literals (Section 4.1.2) are wrapped by objects of type `TimeLiteral.java`. Global from and to selectors (Section 4.1.3) are not wrapped in GDL, but immediately transformed to a set of 'local' ones after parsing the query. As an example, consider the constraint `val_from > Timestamp(2020-01-01)`. Suppose the query graph pattern only contains just two elements `a` and `b`. Then, the constraint would be transformed to

```
a.val_from >  Timestamp(2020-01-01)
AND b.val_from > Timestamp(2020-01-01)
AND MAX(a.val_from, b.val_from) <= MIN(a.val_to, b.val_to).
```

Evidently, comparisons involving MIN and MAX expressions (Section 4.1.4) can be reduced to comparisons between primitive time types, i.e. from/to selectors and time stamps. For example

$$\begin{gathered} \textit{min\{a.val-from, b.val-from\}} < \textit{c.val-from} \\ \Leftrightarrow \\ (\textit{a.val-from} < \textit{c.val-from}) \vee (\textit{b.val-from} < \textit{c.val-from}). \end{gathered}$$

However, such reductions are not applied in GDL. This is because programs using GDL (here Gradoop) should be able to decide how to handle these expressions, i.e. whether to translate them to simple comparisons or not. Thus, MIN and MAX expressions are wrapped by classes `MinTimePoint.java` and `MaxTimePoint.java`.

Interval relations (Section 4.3.2) are only syntactic sugar, as they can be reformulated using only time stamps. Equivalent expressions using only comparisons between time stamps can be obtained from the Tables 4.3, 4.5 and 4.6. Hence, there is no interval type in GDL. Intervals need never be materialized, as they only occur in the context of their relations with other intervals or time stamps. During parsing, these relations are directly translated to the equivalent expressions over time stamps.

Durations of intervals (Section 4.4) are wrapped by `Duration.java`. The constant durations (`Days`,...), however, are wrapped using another `TimePoint.java`, `TimeConstant.java`. The usage of different wrappers for the two types of durations is motivated by the possibility to use constants for other purposes in future work.

All aspects implemented in this thesis have now been discussed. The next Section deals with temporal path expressions and why they have not been implemented.

4.7. Temporal Path Expressions

Temporal constraints on path expressions like `(p1)-[:knows*1..3]->(p2)` are more difficult than on simple edges.

The main reason for this is that the edges on such a path can not be referenced via variable names in a GDL query. Thus, the constraints introduced above are not applicable on path expressions. Because of that, special keywords and/or relations must be defined for these expressions. There is, however, a great variety of possible temporal constraints on path expressions. To illustrate this, consider two examples:

- Suppose, a transportation network is given as a temporal property graph. Vertices represent train stations and every edge corresponds to a connecting train from one station to another. The departure and arrival times of a train are given in the valid interval of the connection edge. To plan a journey from city *c1* to city *c2*, a path expression `(c1)-[:train*1..*]->(c2)` might be used in a GDL query. However, to actually receive reasonable journeys, two temporal constraints must be imposed: first, on stations between *c1* and *c2*, a train must always leave after the train before arrived. Second, the interval between arrival and departure of subsequent trains must not be too long, as a passenger typically does not want to wait for hours for the next train. More formally, if the maximum changing time is one hour and $e_0, ..., e_n$ are the edges on the path between $c1$ and $c2$ in a matching path expression it must hold that:

$$e_i.\textit{val-to} < e_{i+1}.\textit{val-from} \wedge \textit{length}([e_i.\textit{val-to}, e_{i+1}.\textit{val-from}]) \leq 1h$$
$$\forall i \in [0, n-1]$$

  This constraint can not be expressed directly in GDL, as the e_is are not available in the query.

Lukas Christ
3756213

- To give another example, let the graph be a computer network. Here, the vertices are computers and the edges connecting them their connections in the network. Valid times of edges indicate the intervals during which a connection is available. If $c1$ and $c2$ are computers in this network, a connection between them can be expressed as `(c1)-[*1..*]->(c2)`. Suppose, that this connection must be stable during some interval $I = [I_f, I_t]$. Then, the valid intervals of all edges $e_0, ..., e_n$ on the path from $c1$ to $c2$ must contain I, i.e. the constraint here is

  $$e_i.\textit{val-from} \leq I_f \wedge e_i.\textit{val-to} \geq I_t \ \forall i \in [0, .., n]$$

  Different from the transportation network example before, the constraint refers to all edges on the path. Furthermore, the sequence of valid from/to time stamps on the path is irrelevant here, but the edges' valid times must overlap. Similar to the first example, the constraint can not be expressed in GDL as the e_i's are not available either.

The examples make clear that there are many types of temporal constraints for paths: constraints might refer to vertices or edges or both; overlap might be necessary or not; they may impose a sequence on one or more of the four time stamps or not; a sequence of time stamps can be (strictly) ascending or (strictly) descending; there can be duration constraints on vertices/edges, between vertices/edges and so on.

There are two options to integrate temporal path expressions into GDL: one could support just a few types of temporal path expressions, e.g. time-respecting paths [54], stable connections or journeys as shown in the examples. In this case, it would suffice to introduce a few additional keywords corresponding to those constraints. The other option is to enable the user to define arbitrary custom temporal path constraints. To achieve this, however, it would be necessary to conceive a very rich and flexible syntax extension.

Because of the complexity of the temporal path expression problem, it is not addressed in the thesis any further. Instead, the next Section focuses on the processing of GDL queries by Gradoop.

5. Temporal Pattern Matching in Gradoop

Now that temporal queries are possible in GDL, Gradoop must be able to handle them. The matching operator for EPGM graphs was already implemented in Gradoop by Junghanns et al. [28]. In this Section, the approach by Junghanns et al. is described in more detail than in [28]. Furthermore, the extensions necessary to process temporal queries are discussed. Path expressions are not dealt with here, as they are not extended for temporal pattern matching in this thesis.

First, Junghanns et al.'s method is presented in Section 5.1 from a theoretic point of view. Section 5.2 then explains its implementation in Gradoop.

Note that the following only introduces the basic principles of (Temporal) Pattern Matching in Gradoop. Approaches to further optimize the program's runtime are presented in chapter 6.

5.1. Pattern Matching by Relational Query Processing

The processing of a GDL query in Gradoop is conceptually similar to query executions in relational databases. First, a query is translated into relational algebra (Section 5.1.1). Then, a query plan for its execution is constructed. The concept of a query plan is introduced in Section 5.1.2, the construction of such plans in Sections 5.1.3 and 5.1.4. Section 5.1.5 explains how the approach by Junghanns et al. discussed in the Sections 5.1.1-5.1.4 is extended to support the temporal GDL queries introduced in Section 4.

5.1.1. Translation to Relational Algebra

An EPGM graph element can be thought of as a tuple in a relation. To store a whole property graph, two relations $\mathcal{V}$ and $\mathcal{E}$ for vertices and edges respectively, were needed. $\mathcal{V}$ would contain an ID attribute for each vertex as well as an attribute for every property from K. Furthermore, every vertex's label must be stored in $\mathcal{V}$. This attribute is called `__label` in the following, in order not to confuse the label with a property from K. In Table 5.1, a sample of $\mathcal{V}$ for the property graph in Figure 1.1 is given.

ID	__label	Age	Domain	...	Salary
01	Company	NULL	"Insurances"	...	NULL
02	Company	NULL	"Banking"	...	NULL
03	Person	27	NULL	...	NULL
...	...	...	...	...	...
07	University	NULL	NULL	...	NULL

Table 5.1.: Sample from $\mathcal{V}$ of graph in Figure 1.1

The IDs are made up, Gradoop assigns IDs to data graph elements automatically. Information about logical graphs is omitted in this discussion, as it is not vital for the approach.

Lukas Christ
3756213

The relation $\mathcal{E}$ containing edges is more complex, since it additionally needs to contain the source and target vertex ID (`sID` and `tID`) for every edge. Table 5.2 shows a fraction of the edge relation $\mathcal{E}$ for the property graph in Figure 2.1.

ID	sID	tID	__label	Age	Domain	...	Salary
01	03	01	work_at	NULL	NULL	...	5000
02	03	04	knows	NULL	NULL	...	NULL
03	03	06	study_at	NULL	NULL	...	NULL
...	...	...	...	...	...	...	...
10	05	04	knows	NULL	NULL	...	NULL

Table 5.2.: Sample from $\mathcal{E}$ of graph in Figure 1.1

Even though the implementation of the approach will be discussed in detail in Section 5.2, it should be noted already here that this is not the way Gradoop actually stores vertices and edges. Instead, POJOs representing vertices and edges in Gradoop are converted to resemble these relations in the implementation.

Moreover, matches for a query can be represented by a relation in a natural way. All matches for a query have the same structure, especially the same number of vertices and edges. Thus, the attributes of a relation representing query matches are simply the union of attributes of the relations representing the match's single vertices and edges. For example, every result for the query in Figure 2.6 consists of 3 vertices and 2 edges. The underlying data graph (Figure 2.1) has the property set *K={Name, Age, ..., Employees}*. A relational representation of the vertex v would have the attributes `v.ID`, `v.__label`, `v.Name`, `v.Age`,..., `v.Employees`. Analogously, a relational representation of edge e has the attributes `e.ID`, `e.sID`, `e.tID`, `e.__label`, `e.Name`, `e.Age`,..., `e.Employees`. The relation for a query result then unites all these attributes. In the given example it has the attributes `v1.ID`, `v1.__label`, `v1.Name`...,`v1.Employees`, `v2.ID`, `v2.__label`,...,`v2.Employees`, `v3.ID`, `v3.__label`,..., `v3.Employees`, `e1.ID`, `e1.sID`, `e1.tID`, `e1.__label`, `e1.Name`,..., `e1.Employees`, `e2.ID`, `e2.sID`,...,`e2.Employees`. Every tuple of such a relation then represents a match. However, the set of attributes can be more compact. It is sufficient for the relation to only contain the IDs of all vertices and edges. All the property values can be reconstructed with these IDs via joins with $\mathcal{V}$ and $\mathcal{E}$. In the following, the desired result table will be denoted $\mathcal{R}$.

The goal of pattern matching can now be reformulated: finding all matches for a query means to construct $\mathcal{R}$ (representing all correct matches) from the relations $\mathcal{V}$ and $\mathcal{E}$ (representing all vertices and edges of the graph) by means of relational algebra. Put differently, a relational algebra term $\mathfrak{t}$ referring to $\mathcal{V}$ and $\mathcal{E}$ is sought that results in $\mathcal{R}$. Every intermediate result of $\mathfrak{t}$ as well as its end result $\mathcal{R}$ represents a set of sub graphs, each of them a tuple in the corresponding relation.

To determine $\mathfrak{t}$, suitable candidate relations for every vertex and edge in the query graph must first be selected from $\mathcal{V}$ and $\mathcal{E}$. In this first step, only the clauses referring to single elements are used to determine the candidates. More formally, suppose the query vertices are $v_1, ..., v_n$ and $K = \{k_1, ..., k_m\}$. Then, candidate relations $cand(v_1)...cand(v_n)$ are computed as stated in Equation 5.1.

$$cand(v_i) = \rho_{v_i.k_1/k_1}\ \rho_{v_i.k_2/k_2}\ \cdots\ \rho_{v_i.k_m/k_m}\ \rho_{v_i.ID/ID}(\Pi_{K\cup\{ID\}}(\sigma_{\theta_{\{v_i\}}}(\mathcal{V}))) \tag{5.1}$$

All rename operations (ρ) here only ensure that all attribute names in $cand(v_i)$ are prefixed with $v_i.$. This will enable joins between candidate relations later. The projection (Π) excludes the *___label* attribute, as it is not needed further (labels can not be compared in GDL yet). The selection (σ) checks whether all constraints referring to the query vertex in question ($\theta_{\{v_i\}}$) are satisfied.

For edges, the candidate selection is a bit more complicated. Every edge e_i has a source vertex v_j and a target vertex v_k. Their IDs are stored in columns *e.sID* and *e.tID* respectively. In $cand(e_i)$, these columns are renamed $v_j.ID$ and $v_k.ID$ immediately, in order to enable natural joins with $cand(v_j)$ and $cand(v_k)$ later on. The label attribute can be discarded immediately, same as for vertices. Equation 5.2 defines edge selection formally for an edge $e_i = (v_j, v_k)$.

$$cand(e_i) = \rho_{e_i.k_1/k_1}\cdots\ \rho_{e_i.k_m/k_m}\ \rho_{v_j.ID/e_i.sID}\ \rho_{v_k.ID/e_i.tID}\ \rho_{e_i.ID/ID}(\Pi_{K\cup\{ID,e_i.sID,e_i.tID\}}\sigma_{\theta_{\{e_i\}}}(\mathcal{E})) \tag{5.2}$$

Every relation $cand(v_i)$ now comprises a set of vertices potentially matching the query vertex v_i. Analogously, $cand(e_i)$ is the set of potential matches for the query edge e_i. At this point, it is already ensured that all constraints referring to only one element are checked, i.e. fulfilled in every result. In the next steps, all candidate relations are joined, yielding $\mathcal{R}$. During these joins, the remaining constraints on more than one element are checked. Properties that are not needed for any further joins are discarded via projections. Such projections are noted $project(\mathfrak{t}', \theta)$, meaning that an attribute in $\mathfrak{t}'$ is kept if and only if it is referenced in θ. Discarding properties as early as possible keeps intermediate results during the execution of the term small, which affects the run time positively.

Algorithm 1 formalizes this discussion. It assumes the homomorphism paradigm and results in the desired relational algebra term $\mathfrak{t}$, which yields $\mathcal{R}$.

Algorithm 1 Simple Translation of a query to Relational Algebra (Homomorphism)

```
1:  procedure SimpleQueryToRel(((V_q, E_q), θ))
2:      terms = ∅
3:      for v ∈ V_q do
4:          terms = terms ∪ cand(v) (Eq. 5.1)
5:          θ = θ\θ_{v}
6:      for e ∈ E_q do
7:          terms = terms ∪ cand(e) (Eq. 5.2)
8:          θ = θ\θ_{e}
9:      while |terms| > 1 do
10:         t_1, t_2 = choose two random terms from terms
11:         naturalJoinVars = GetAllVars(t_1) ∩ GetAllVars(t_2)
12:         processedVars = GetVars(t_1) ∪ GetVars(t_2)
13:         naturalJoinPredicates = NJPredicates(naturalJoinVars)
14:         joinPredicate = θ_{processedVars} ∧ naturalJoinPredicates
15:         new_term = empty relational algebra term
16:         if joinPredicate empty then
17:             new_term = t_1 × t_2
18:         else
19:             new_term = project(t_1 ⋈_{joinPredicate} t_2, θ\θ_{allVars})
20:         terms = (terms\{t_1, t_2}) ∪ {new_term}
21:         θ = θ\ θ_{allVars}
22:     t = only remaining term in terms
23:     return t
24:
25: procedure GetVars(t)
26:     if t of the form cand(x), x vertex or edge then
27:         return {x}
28:     else if t of the form t_1 ⋈_{θ'} t_2 or t_1 × t_2 then
29:         return GetVars(t_1) ∪ GetVars(t_2)
30:
31: procedure GetAllVars(t)
32:     if t of the form cand(v), v vertex then
33:         return {v}
34:     else if t of the form cand(e), e = (s, t) edge then
35:         return {e, s, t}
36:     else if t of the form t_1 ⋈_{θ'} t_2 or t_1 × t_2 then
37:         return GetAllVars(t_1) ∪ GetAllVars(t_2)
```

In line 12, a function NJPredicates is used. This function computes natural join predicates of the form $x.ID = x.ID$ for every variable x that is referenced in the results of both terms t_1 and t_2. The distinction between GetVars and GetAllVars is motivated by the structure of edge relations, which include source and target vertex IDs. Results of GetAllVars include source and target vertices of edges in order to enable joins on them (line 11, 13), while results of GetVars do not.

Figure 5.1 illustrates a possible run of Algorithm 1 for the example query in Figure 2.6. Assume that the *company* vertices have the IDs `01` and `02` and the vertices of Alice, Bob and Carl the IDs

03, 04 and 05. The *work_at* edges have the IDs 01-04 (from left to right in Figure 2.2). Projections are ommitted for the sake of brevity.

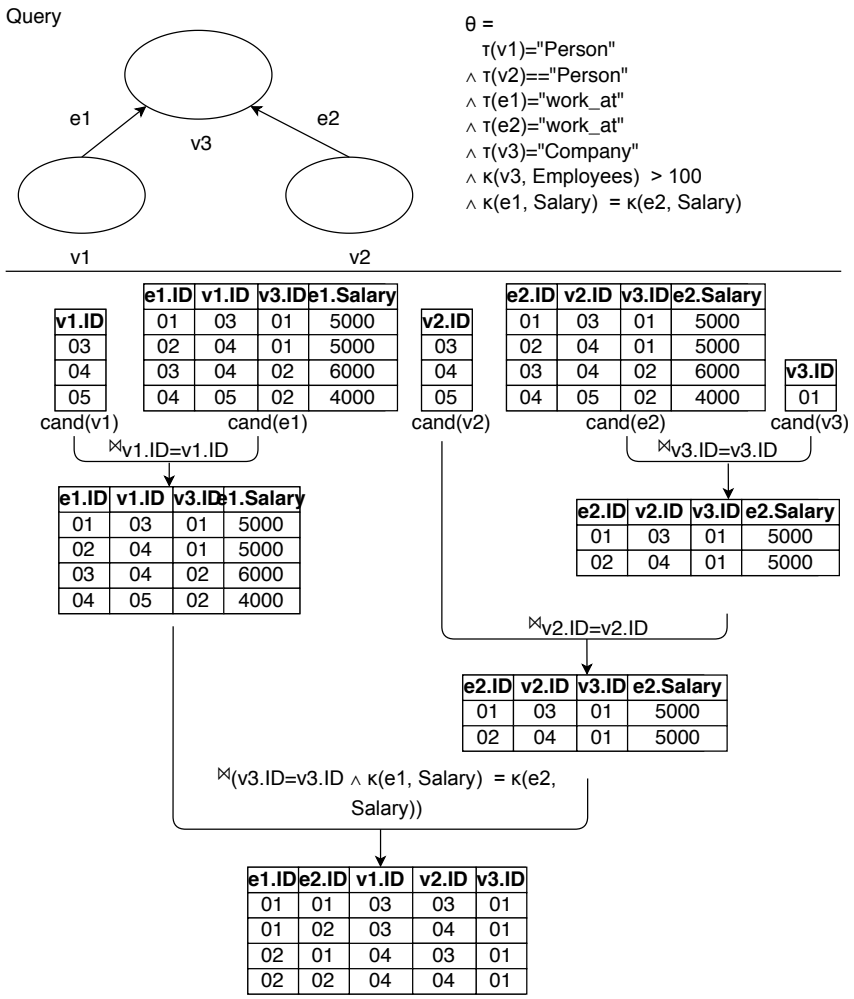

Figure 5.1.: Exemplary run of Algorithm 1

Lemma 1. *For any query,* $((V_q, E_q), \theta)$*, Algorithm 1 computes a relational algebra expression* $\mathfrak{t}$ *that results in the correct result relation* $\mathcal{R}$ *for the homomorphism matching paradigm.*

Proof. Algorithm 1 terminates for every query, as every time the loop (lines 9-21) is executed, $|terms|$ is reduced by 1 (line 20).

Furthermore, it is clear, that the structure of $\mathcal{R}$'s results is as desired: For every query graph element, the respective candidates are selected. These candidates are then subsequently joined until no more join is possible. Lines 11, 13 and 14, combined with edge selections (Equation 5.2, line 7) ensure that the correct structure is created, i.e. every edge is joined with its correct source and target vertex.

Moreover, all predicates specified in θ must be fulfilled for every resulting embedding. For a query graph element x, the constraints $\theta_{\{x\}}$ are used as selection predicates to filter the candidates for x (lines 4, 7). Every other sub-CNF of θ is a θ_X for $X \subseteq (V_q \cup E_q)$. Lines 14, 16 and 19 make sure

that θ_X is checked as soon as a term comprising all variables in X exists in $terms$. This will always be the case during Algorithm 1, as it terminates and $\mathfrak{t}$ is a term containing all variables. □

Note that for the isomorphism paradigm, an additional operation would be needed. This operation would remove tuples that do not fulfill the isomorphism condition, i.e. tuples containing the same graph element more than once. Obviously, this operation could easily be integrated in the Algorithm given here (it is in fact implemented in Gradoop). To keep the discussion simple, however, this aspect is omitted in the following. Instead, the query paradigm is assumed to be homomorphism throughout the following discussion.

Algorithm 1 describes a very simple procedure of 'translating' a query into relational algebra. This is not the algorithm used in Gradoop. It rather serves as an introduction to Junghanns et al.'s approach [28], which is a more refined version of it.

A major problem of Algorithm 1 is that the choice of the next join is random in line 10. However, this choice is not arbitrary, as the join sequence influences the size of intermediate results and thus the run time of the algorithm. Consider the following example. Let $q = ((\{v_1, v_2\}, \{e = (v_1, v_2)\}), \theta = \theta_{\{v_1\}} \wedge \theta_{\{e\}} \wedge \theta_{\{v_2\}})$ be a simple query. Suppose that there are 10 matches for v_1, 20 for e and 50 for v_2, i.e. $|cand(v_1)| = 10$, $|cand(e)| = 20$, $|cand(v_2)| = 50$. Furthermore, suppose that every candidate for v_1 matches exactly one edge candidate, while 15 candidates for v_2 match exactly one edge candidate and the other 35 candidates for v_2 do not match any edge. Let the overall result size be 5. There are three possible join sequences that may be produced for q by Algorithm 1. Table 5.3 compares them w.r.t. the size of the intermediate results when they are executed.

Join Sequence	Result sizes before 1^{st} join	Result sizes before 2^{nd} join	Final result size
$(c(v_1) \times c(v_2)) \bowtie c(e)$	10, 20, 50	500, 20	5
$(c(v_1) \bowtie (c(e)) \bowtie c(v_2)$	10, 20, 50	10, 50	5
$c(v_1) \bowtie (c(e) \bowtie c(v_2))$	10, 20, 50	10, 15	5

Table 5.3.: Exemplary join sequences and their (intermediate) cardinalities

Of course, the candidate sets and the final results are always of equal size. However, the intermediate results' cardinalities differ. The third join sequence is arguably the most efficient one to process, as it minimizes the intermediate results' sizes. Hence, it is not reasonable to create the join sequence randomly, as done in line 10 of Algorithm 1.

Determining the best join sequence is not trivial, as the size of intermediate results is not known in advance. In addition, there are $\Pi_{i=2}^{n}\binom{i}{2}$ possible sequences for n query graph elements.

Junghanns et al. [28] employ a query planning approach in order to find a good join sequence. Its basic idea is to the create the relational algebra expression in a greedy manner. In every step the next join is chosen that keeps the expected intermediate results as small as possible. The size of the intermediate result can only be estimated at this point, as it can only be determined by actually executing the (sub-)query.

In what follows, the concept of a query plan is introduced (Section 5.1.2), which is a prerequisite for the refined version of Algorithm 1. This algorithm is then presented in Section 5.1.3, followed by the explication of the necessary estimation of intermediate results sizes in Section 5.1.4.

5.1.2. Query Plans

Query plans are the basis of Gradoop's matching operator. A query plan represents a relational algebra term with an unambiguous join sequence. Consider such a relational algebra term, e.g. the resulting term in Figure 5.1. As Figure 5.1 shows, it can be depicted hierarchically, yielding a binary tree whose nodes are associated with relational algebra expressions. Query plans are basically such trees, only that the nodes in a query plan are augmented with further information about the term represented by them. This information is used to construct the whole query plan, aiming at an efficient join sequence. Having done that, the query plan's nodes can be mapped to actual 'physical' operators in order to execute the query in the implementation. The nodes of a query plan are called *plan nodes* henceforth.

Definition 19 (Plan Node)**.** *A* plan node n *is a node in a query plan (Def. 27). It is associated with five aspects:*

- $\mathfrak{t}_n$ *is the relational algebra expression represented by* n
- $var(n)$ *denotes the set of variables occurring in* $\mathfrak{t}_n$ *for which all properties relevant for further processing are included in* $\mathfrak{t}_n$*'s result.*
- $all\_vars(n)$ *represents the set of all variables that are present in attributes of the result of* $\mathfrak{t}_n$*. It always holds that* $var(n) \subseteq all\_vars(n)$
- $\theta^{(n)}$ *is the predicate containing all predicates of* $\mathfrak{t}_n$*, i.e. all predicates checked by* n
- $children(n)$ *denotes the set of* n*'s children, which are plan nodes themselves*
- $cardinality(n) \in \mathbb{N}$ *is the estimated cardinality of* $\mathfrak{t}_n$*'s result*

Plan nodes generally correspond to operations of relational algebra used in Algorithm 1.

The distinction between $var(n)$ and $all\_vars(n)$ is due to edge selections. They result in relations containing 3 variables overall (edge, source vertex, target vertex) but are only guaranteed to include the relevant properties for the edge variable.

There are seven types of plan nodes, as defined below. The estimation of the plan nodes' respective *cardinality* value is discussed in Section 5.1.4 and thus not treated in these definitions.

Selections are represented differently for vertices and edges, because edge selections are special as pointed out above. Thus, there are two different node types for selections.

Definition 20 (Vertex Selection Node)**.** *A* vertex selection node *n is a plan node that represents a single, initial vertex candidate selection expression (Equation 5.1). This means that $\mathtt{t}_n = cand(v)$ and $var(n) = \{v\}$. It follows that $\theta^{(n)} = \theta_{\{v\}}$. The vertex selection yields a relation referring only to v, hence $all\_vars(n) = var(n) = \{v\}$. A vertex selection node is always a leaf node, i.e. an atom of the relational algebra expressions used here. Hence, $children(n) = \emptyset$.*

Definition 21 (Edge Selection Node)**.** *An* edge selection node *n is a plan node that represents a single, initial edge candidate selection expression (Equation 5.2). Thus, $\mathtt{t}_n = cand(e)$ and $var(n) = \{e\}$ and $\theta^{(n)} = \theta_{\{e\}}$. The edge selection yields a relation with attributes referring to 3 different elements (edge, source and target vertex), so $|all\_vars(n)| = 3$. Like a vertex selection node, an edge selection node is always a leaf node, hence $children(n) = \emptyset$.*

Due to implementation aspects discussed later, there are three types of join nodes: a cartesian product type (Def. 22), one for equijoins on edge source/target IDs (Def. 23) and one for equijoins on property values from K (Def. 24). Joins with join predicates other than equality are represented as a combination of an appropriate join node and a filter node (Def. 25).

Definition 22 (Cartesian product Node)**.** *A* cartesian product node *n is a plan node that represents a cartesian product of two relations. Therefore, $children(n) = \{c_1, c_2\}$ for $\mathtt{t}_n = c_1 \times c_2$. Furthermore, $var(n) = var(c_1) \cup var(c_2)$ $(var(c_1) \cap var(c_2) = \emptyset)$, $all\_vars(n) = all\_vars(c_1) \cup all\_vars(c_2)$ and $\theta^{(n)} = \theta^{(c_1)} \wedge \theta^{(c_2)}$.*

Definition 23 (Edge Join Node)**.** *An* edge join node *n is a plan node that represents a join on an edge's source or target vertex ID. It is obviously always a binary node, so that $children(n) = \{c_1, c_2\}$. The represented relational algebra term is $\mathtt{t}_n = c_1 \bowtie_\phi c_2$. Here, ϕ is a natural join predicate of the form* x.ID=x.ID*, where x is a source and/or target vertex of some edge e ($e \in var(c_1) \vee e \in var(c_2)$). Furthermore, $var(n) = var(c_1) \cup var(c_2)$, $all\_vars(n) = all\_vars(c_1) \cup all\_vars(c_2)$ and $\theta^{(n)} = \theta^{(c_1)} \wedge \theta^{(c_2)} \wedge \phi$*

Definition 24 (Value Join Node)**.** *A* value join node *n is a plan node representing an equijoin on property values. It is always a binary node with $children(n) = \{c_1, c_2\}$. The represented relational algebra term is $\mathtt{t}_n = c_1 \bowtie_\phi c_2$, where ϕ is an equijoin on property values of variables from $var(n)$. Moreover, $var(n) = var(c_1) \cup var(c_2)$, $all\_vars(n) = all\_vars(c_1) \cup all\_vars(c_2)$ and $\theta^{(n)} = \theta^{(c_1)} \wedge \theta^{(c_2)} \wedge \phi$.*

Definition 25 (Filter Node)**.** *A* filter node *n is a plan node that represents a selection operator (σ). It has one binary child node c, thus $children(n) = \{c\}$. The represented relational algebra term is $\mathtt{t}_n = \sigma_\phi(\mathtt{t}_c)$, where ϕ is a predicate on properties including only variables from $var(c)$. Hence, $var(n) = var(c)$, $all\_vars(n) = all\_vars(c)$ and $\theta^{(n)} = \theta^{(c)} \wedge \phi$.*

Moreover, property projections are represented by a *projection node*.

Definition 26 (Projection Node)**.** *A* projection node n *is a plan node that represents a projection. It has one binary child node* c*, thus* $children(n) = \{c\}$*. The represented relational algebra term is* $\mathfrak{t}_n = \Pi_{k_1,\ldots,k_l}(\mathfrak{t}_c)$ *where* $k_1 \ldots k_l$ *are properties present in the result of* $\mathfrak{t}_c$*. As no variables and predicates are added compared to the only child* c*,* $var(n) = var(c)$*,* $all\_vars(n) = all\_vars(c)$ *and* $\theta^{(n)} = \theta^{(c)}$*.*

Now, the translation from GDL to relation algebra can be thought of as the construction of a *query plan.*

Definition 27 (Query Plan)**.** *A* query plan p *is a binary tree constructed from plan nodes. Like for plan nodes,* $\mathfrak{t}_p$*,* $var(p)$*,* $all\_vars(p)$ *and* $\theta^{(p)}$ *are defined. Let* r *be the root node of* p*. Then,* $\mathfrak{t}_p = \mathfrak{t}_r$*,* $var(p) = var(r)$*,* $all\_vars(p) = all\_vars(r)$ *and* $\theta^{(p)} = \theta^{(r)}$*.*

A query plan p with root r evidently represents the unambiguous relational algebra term $\mathfrak{t}_r$. All leaf nodes are vertex or edge selection nodes that select candidates for every variable from the database. Figure 5.2 shows the query plan for an example query with made-up cardinalities.

Conversely, for a query, there is (in general) more than one possible translation to a query plan, as there are (in general) several possible join sequences. If $\mathfrak{p}$ is a query plan for a query $q = ((V_q, E_q), \theta)$, $\theta^{(\mathfrak{p})} = \theta$, $var(p) = V_q \cup E_q$. As already demonstrated, the join sequence (and hence, the query plan) for a query is not arbitrary, but influences the run time. It is thus necessary to construct a query plan that leads to small intermediate results. The next Section deals with this task.

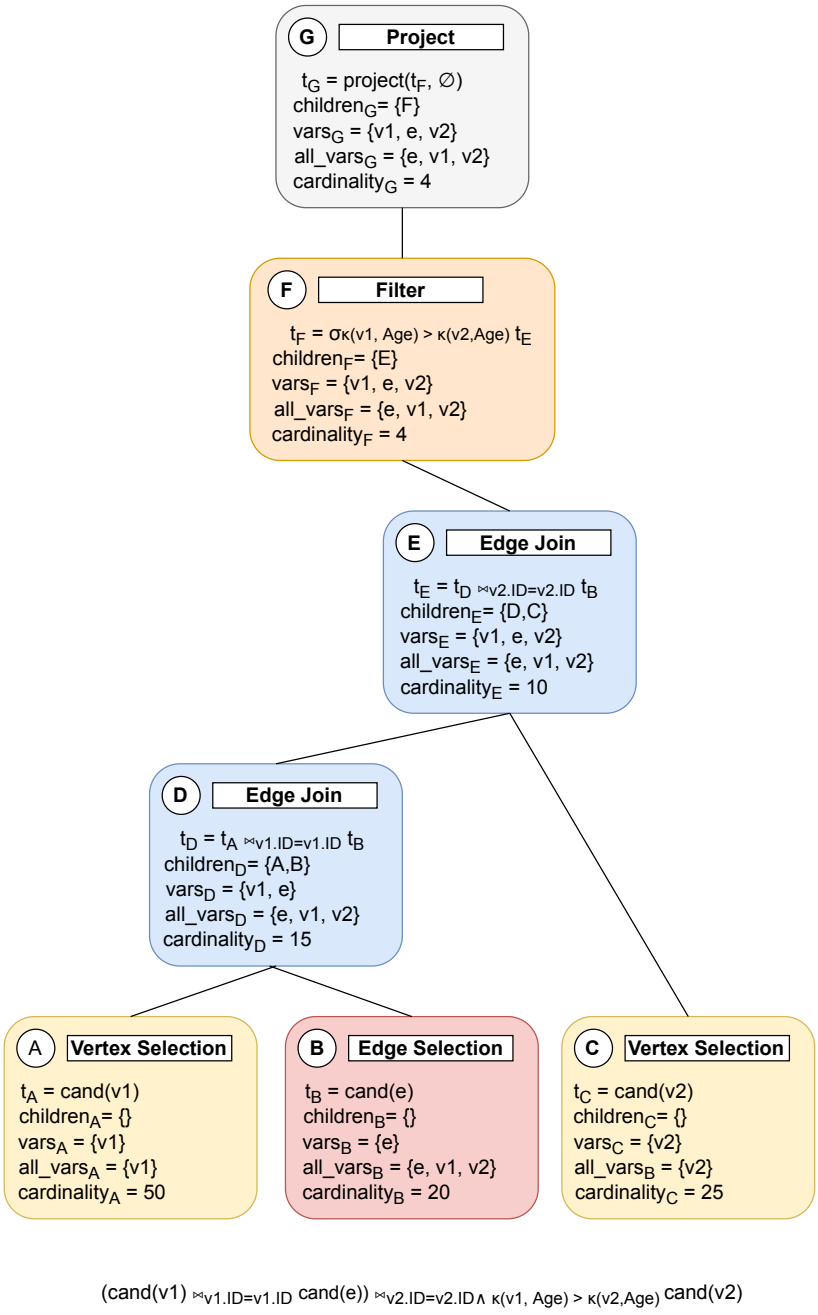

Figure 5.2.: Exemplary query plan

5.1.3. Greedy Query Planning

The construction of a query plan is also referred to as *query planning*. Junghanns et al. [28] employ a greedy method for query planning in Gradoop: the query plan is constructed incrementally, in a bottom-up manner. The algorithm introduced here can be seen as a refined version of Algorithm 1, as already mentioned. Instead of choosing the next join randomly, it is chosen according to a cost/cardinality estimation. Algorithm 2 presents the general approach by Junghanns et al. Assume that a query plan is created by a constructor $QueryPlan(node)$ where *node* is the root node. Vertex/edge selection nodes are initialized with their corresponding variable.

Algorithm 2 Translation of a query to a query plan (Homomorphism)

```
1:  procedure QueryToQueryPlan(((V_q, E_q), θ))
2:      plans = ∅
3:      for v ∈ V_q do
4:          plans = plans ∪ {QueryPlan(VertexSelectionNode(v))}
5:      for e ∈ E_q do
6:          plans = plans ∪ {QueryPlan(EdgeSelectionNode(e))}
7:      while |plans| > 1 do
8:          new_plans = EvalEdgeJoins(plans)
9:          if new_plans = ∅ then
10:             new_plans = EvalOtherJoins(plans, θ)
11:         new_plans = EvalFilters(new_plans)
12:         new_plans = EvalProjections(new_plans)
13:         p = argmin_{p' ∈ new_plans} EstimateCardinality(p')
14:         plans = plans ∪ {p}
15:         for q ∈ plans do
16:             if var(q) ⊆ var(p) then
17:                 plans = plans \ {q}
18:     p = only remaining plan in plans
19:     return p
```

Algorithm 2's structure closely resembles the approach presented in Algorithm 1. Instead of terms, a set of query plans (*plans*) is managed. Initially, for each query vertex and edge a corresponding plan, consisting only of one vertex/edge selection node (lines 3-6) is created. Then, joins are built from these plans (lines 8-10) until only one plan is left. Joining two plans with two roots r_1 and r_2 here means creating a plan with root r and $children(r) = \{r_1, r_2\}$. Edge joins, i.e. joins on edges and their source/target vertices, are always preferred (line 8). Only if no such join is possible, other types of joins are taken into consideration (lines 9, 10). The next join is not chosen randomly anymore, but determined by some function EstimateCardinality in line 13.

In the following, the 4 functions called in lines 8-13 of Algorithm 2 are discussed. Algorithm 3 shows the function used to determine possible natural joins on the set of plans (line 8 in Algorithm 2). Non-leaf plan nodes are always created using constructors that take their child(ren) as argument(s) in what follows.

Algorithm 3 Evaluate possible natural joins on a set of plans

```
1:  procedure EvaluateEdgeJoins(plans)
2:      new_plans = ∅
3:      for p_1 ∈ plans do
4:          for p_2 ∈ plans, p_1 ≠ p_2 do
5:              if all_vars(p_1) ∩ all_vars(p)_2 ≠ ∅ then
6:                  new_plans = new_plans ∪
                        {QueryPlan(EdgeJoinNode(p_1, p_2))}
7:      return new_plans
```

Edge joins are in most cases more selective than other joins because for every edge candidate, there is only one vertex candidate that satisfies the condition of such a join. If no edge join is applicable,

value joins and eventually cartesian products are considered in line 10 using Algorithm 4. Here, $equi(\theta)$ refers to the CNF of all equality comparisons in the CNF θ.

Algorithm 4 Evaluate possible value joins and cartesian products on a set of plans

```
1:  procedure EvaluateOtherJoins(plans, θ)
2:      new_plans = ∅
3:      for 𝔭1 ∈ plans do
4:          for 𝔭2 ∈ plans, 𝔭1 ≠ 𝔭2 do
5:              new_plan = NULL
6:              if equi(θ_{var(𝔭1)∪var(𝔭2)} \ (θ^(𝔭1) ∧ θ^(𝔭2))) ≠ ∅ then
7:                  new_plan = QueryPlan(ValueJoinNode(𝔭1, 𝔭2)))
8:              else
9:                  new_plan = QueryPlan(CartesianProductNode(𝔭1, 𝔭2))
10:             new_plans = new_plans ∪ {new_plan}
11:     return new_plans
```

If a value join is applicable for two plans, it is preferred over a cartesian product. This is reasonable, as a value join will reduce the size of intermediate results directly.

In every iteration of the loop in Algorithm 2, after the execution of line 10, *new_plans* contains at least one plan. Because Algorithms 3 and 4 only allow natural or equi-joins as join predicates, it must be determined whether other predicates can be checked on the variables of each plan in *new_plans*. This is done by Algorithm 5 in line 11 of Algorithm 2.

Algorithm 5 Evaluate possible natural joins on a set of plans

```
1:  procedure EvalFilters(plans, θ)
2:      new_plans = ∅
3:      for 𝔭 ∈ plans do
4:          new_plan = NULL
5:          if (θ_{var(𝔭)} \ θ^(𝔭)) ≠ ∅ then
6:              new_plan = QueryPlan(FilterNode(𝔭))
7:          else
8:              new_plan = 𝔭
9:          new_plans = (new_plans \ {𝔭}) ∪ {new_plan}
10:     return new_plans
```

Evaluating the plan set for application of further selections, i.e. filters, is simple. For every plan $\mathfrak{p}$ in *plans*, it is checked whether there are predicates in θ that could be evaluated on $var(\mathfrak{p})$, but aren't yet associated with a node in $\mathfrak{p}$ (line 5). This is evidently the case if and only if $\theta^{(\mathfrak{p})} \subsetneq \theta_{var(\mathfrak{p})}$ Only if such predicates are found, a new filter node is created for them and made the new root of $\mathfrak{p}$ (line 6-9). Then, for the new version of $\mathfrak{p}$, $\theta^{(\mathfrak{p})} = \theta_{var(\mathfrak{p})}$.

After a selection is applied to a relation, properties can be deleted if they are not referenced in the remaining predicates. Thus, in line 12 of Algorithm 2, every plan in *plans* is checked for the applicability of a projection operation. All properties that are not referred to in $\theta \backslash \theta^{(\mathfrak{p})}$ can be deleted from $\mathfrak{t}_{\mathfrak{p}}$ by making a corresponding projection node the new root of $\mathfrak{p}$.

If there is more than one new plan in *new_plans*, estimations of the cardinalities are employed to greedily decide which plan is chosen as the next partly solution (line 13 in Algorithm 2). The estimation function will be discussed in Section 5.1.4.

Lemma 2. *Algorithm 2 is correct, i.e. it yields for any query* $q = ((V_q, E_q), \theta)$ *a query plan* $\mathfrak{p}$ *where* $\mathfrak{t}_\mathfrak{p}$ *returns the correct result table* $\mathcal{R}$*.*

Proof. First, it must be shown that at the beginning of the loop (line 8), for $\mathfrak{p}, \mathfrak{q} \in plans$ $(\mathfrak{p} \neq \mathfrak{q})$, $var(\mathfrak{p}) \cap var(\mathfrak{q}) = \emptyset$. This is clear for the first iteration of the loop, as initially, there are only plans containing exactly one vertex/edge selection node (lines 3-6). Joining them, however, never changes this fact: For every type of binary node n with children c_1, c_2, $var(n) = var(c_1) \cup var(c_2)$. Filters and Projections of any n do not impact $var(n)$. As soon as a new plan $\mathfrak{p}$ is created from $\mathfrak{p}_1$ and $\mathfrak{p}_2$, it is added to *plans* (line 14). Then, $\mathfrak{p}_1$ and $\mathfrak{p}_2$ (and only those two) are deleted from *plans* (lines 15-17).

It is thus clear that the algorithm terminates and yields a query plan $\mathfrak{p}$ with $var(\mathfrak{p}) = V_q \cup E_q$.

Let $\mathcal{R}$ be the relation resulting from $\mathfrak{t}_\mathfrak{p}$ The correctness of $\mathcal{R}'s$ structure can be deduced from the use of edge join nodes. It is guaranteed, that every edge is joined with its source and target vertices by lines 6 and 8 and the obvious correctness of Algorithm 3.

Furthermore, every constraint in θ is contained in at least one node n's $\theta^{(n)}$. The application of Algorithm 5 in line 11 and the fact that $var(\mathfrak{p}) = V_q \cup E_q$ imply $\theta^{(\mathfrak{p})} = \theta$.

Overall, the resulting query plan corresponds to a relational algebra term that yields a relation of structurally correct matches for (V_q, E_q) with all constraints in θ being checked in the term. □

Next, Junghanns et al.'s approach to join cardinality estimation (used in line 13 of Algorithm 2) is explained.

5.1.4. Join Cardinality Estimations

Junghanns et al. [28] use methods from the field of query planning in relational databases to compute cardinality estimations. These methods were taken from [19] and are presented in the following.

For the following discussion, let $\mathfrak{p}$ be an arbitrary query plan. Estimating the result set cardinality of $\mathfrak{t}_\mathfrak{p}$ is based on two aspects:

- *joinCardinality*$(\mathfrak{p})$, the estimated result size if all the relations referenced by $\mathfrak{t}_\mathfrak{p}$ were simply joined without considering $\theta^{(\mathfrak{p})}$.
- $sel(\theta^{(\mathfrak{p})})$, the selectivity of the predicates $\theta^{(\mathfrak{p})}$, i.e. an estimation on the percentage of tuples from *joinCardinality*$(\mathfrak{p})$ that are filtered out by the constraints θ.

With these two terms, the cardinality of $\mathfrak{t}_\mathfrak{p}$'s result can simply be estimated as

$$cardinality(\mathfrak{p}) = joinCardinality(\mathfrak{p}) * sel(\theta^{(\mathfrak{p})}) \tag{5.3}$$

The root node of $\mathfrak{p}$ is called r in the following. Hence, by definition,
$cardinality(\mathfrak{p}) = cardinality(r)$ so that cardinalities of plans can be determined using cardinalities of plan nodes, as $joinCardinality(\mathfrak{p}) = joinCardinality(r)$ and $\theta^{(\mathfrak{p})} = \theta^{(r)}$.

For reasons that will become apparent soon, it makes sense to include information about the query element's labels when computing $joinCardinality(r)$. Thus, $\theta^{(r)}$ must be split into constraints referring to labels and such that do not. This means that $\theta^{(r)} = \theta^{(r)\prime} \wedge \theta_\tau^{(r)}$, where $\theta_\tau^{(r)}$ includes a constraint from $\theta^{(r)}$ if and only if it is of the form $x.\_\_label = s$ for some query graph element x and a string s. Conversely, $\theta^{(r)\prime}$ contains all constraints from $\theta^{(r)}$ that are not included in $\theta_\tau^{(r)}$. Then, Equation 5.3 can be reformulated:

$$cardinality(\mathfrak{p}) = cardinality(r) = joinCardinality(r, \theta_\tau^{(r)}) * sel(\theta^{(r)\prime}) \tag{5.4}$$

It must now be examined how to compute $joinCardinality(r, \theta_\tau^{(r)})$ and $sel(\theta^{(r)\prime})$. The second aspect, predicate selectivity, has not been implemented by Junghanns et al. However, in this thesis, an implementation for estimating selectivities is provided. Since it belongs to the efforts to speed up the existing Pattern Matching implementation, it will be discussed within the 'Optimization' part, in Section 6.2. For now, the reader may assume that it is possible to compute a selectivity function $sel(\theta^{(r)\prime})$ for any predicate $\theta^{(r)\prime}$ ($0 \leq sel(\theta^{(r)\prime}) \leq 1$). If $sel(\theta^{(r)\prime}) = 1$, $\theta^{(r)\prime}$ is expected not to filter out any tuple, i.e. the predicate is assumed to be always true. A selectivity estimation close to 0 indicates a highly selective predicate that only evaluates to true on very few, if any, tuples.

Other than selectivity estimation, estimations for $joinCardinality(r, \theta_\tau^{(r)})$ have been implemented by Junghanns et al. [28]. Before the cardinality can be estimated for any join node, the cardinalities for the leaf nodes must be known. As pointed out before, leaf nodes of query plans represent vertex and edge selections.

Gradoop provides methods to compute graph statistics about the number of all vertices and edges as well as the number of vertices/edges with a certain label. In the following, $|V_{label}|$ denotes the number of vertices with label *label*. The notation $\tau(a)$ is used to denote the label of some query graph element a. $|V|$ is the number of all vertices in the graph. Analogously, $|E_{label}|$ and $|E|$ are defined for edges.

Let m be a vertex selection node with $var(m) = \{a\}$. Using basic graph statistics,
$joinCardinality(m, \theta_\tau^{(m)})$ can be computed as

$$joinCardinality(m,\ \theta_\tau^{(m)}) = \begin{cases} |V_{\tau(a)}|, & \text{if } \tau(a) \text{ is specified by } \theta_\tau^{(m)} \\ |V| & \text{otherwise} \end{cases} \tag{5.5}$$

Similarly, for an edge selection node n with $var(n) = \{e\}$,

$$joinCardinality(n,\theta_\tau^{(n)}) = \begin{cases} |E_{\tau(e)}| & \text{if } \tau(e) \text{ is specified by } \theta_\tau^{(n)} \\ |E|, & \text{otherwise} \end{cases}$$

(5.6)

The estimations for leaf cardinalities serve as the base case for the recursive estimation of non-leaf node cardinalities.

Projection and filter nodes obviously do not influence the *joinCardinality* value of the plan they are part of. Hence, if n is a projection or filter node with $children(n) = \{c\}$,

$$joinCardinality(n, \theta_\tau^{(n)}) = joinCardinality(c, \theta_\tau^{(c)}) \tag{5.7}$$

A cartesian join node n with $children(n) = \{c_1, c_2\}$ represents a cartesian product and does not join on any attribute. Its cardinality can hence be simply estimated as

$$joinCardinality(n, \theta_\tau^{(n)}) = joinCardinality(c_1, \theta_\tau^{(c_1)}) * joinCardinality(c_2, \theta_\tau^{(c_2)}) \tag{5.8}$$

Value join nodes can be treated like cartesian join nodes here, as their join predicate is not relevant for *joinCardinality* but taken care of in the estimation of $sel(\theta^{(m)\prime})$, when m is the root node of the query plan containing the value join. Hence, Equation 5.8 also applies to value join nodes.

The estimation of an edge join node is more complicated. Let n be an edge join node with $children(n) = \{c_1, c_2\}$ in the following. Edge join nodes are natural joins, as already pointed out. Garcia-Molina et al. [19] describe a method to estimate the size of natural joins. This method is adapted by Junghanns et al. [28] in order to handle edge join nodes.

In what follows, let $\mathcal{R}_1$, $\mathcal{R}_2$ be the relations to join, i.e. the results of $\mathfrak{t}_{c_1}$ and $\mathfrak{t}_{c_2}$. Let a be the join variable, so that the edge join discussed here is $\mathcal{R}_1 \bowtie_{a.ID=a.ID} \mathcal{R}_2$.

Furthermore, $Values(\mathcal{R}, a.ID)$ denotes the set of distinct values (i.e. candidates) for vertex a's ID in a relation $\mathcal{R}$. The notation $|\mathcal{R}|$ is used to refer to the number of (partial) matches, i.e. tuples in a relation $\mathcal{R}$. P denotes the probability function.

Garcia-Molina et al. [19] make two simplifying assumptions:

- *Containment of Values*: For relations $\mathcal{R}_1$, $\mathcal{R}_2$ with $|Values(\mathcal{R}_1, a.ID)| \leq |Values(\mathcal{R}_2, a.ID)|$, it holds that

$$a' \in Values(\mathcal{R}_1, a.ID) \Rightarrow a' \in Values(\mathcal{R}_2, a.ID)$$

- *Preservation of Values*: For an edge join $\mathcal{R}_1 \bowtie_{a.ID=a.ID} \mathcal{R}_2$ and any property, e.g. $b.ID$, which occurs in $\mathcal{R}_1$, but not in $\mathcal{R}_2$,

$$Values(\mathcal{R}_1 \bowtie_{a.ID=a.ID} \mathcal{R}_2,\ b.ID) = Values(\mathcal{R}_1, b.ID)$$

In order to estimate the cardinality of $\mathcal{R}_1 \bowtie_{a.ID=a.ID} \mathcal{R}_2$, the probability $P(\mathcal{R}_1[i].a.ID = \mathcal{R}_2[j].a.ID)$ must be determined first, where i and j point to arbitrary tuples ($1 \leq i \leq |\mathcal{R}_1|$, $1 \leq j \leq |\mathcal{R}_2|$) in the relations. If $|\mathit{Values}(\mathcal{R}_1, a.ID)| < |\mathit{Values}(\mathcal{R}_2, a.ID)|$, the Containment of Values assumption leads to the estimation

$$P(\mathcal{R}_1[i].a.ID = \mathcal{R}_2[j].a.ID) = \frac{1}{|\mathit{Values}(\mathcal{R}_2, a.ID)|} \tag{5.9}$$

because every ID value in $\mathit{Values}(\mathcal{R}_1, a.ID)$ is also contained in $\mathit{Values}(\mathcal{R}_2, a.ID)$. Analogously, if $|\mathit{Values}(\mathcal{R}_1, a.ID)| < |\mathit{Values}(\mathcal{R}_2, a.ID)|$

$$P(\mathcal{R}_1[i].a.ID = \mathcal{R}_2[j].a.ID) = \frac{1}{|\mathit{Values}(\mathcal{R}_1, a.ID)|} \tag{5.10}$$

Putting both cases together yields

$$P(\mathcal{R}_1[i].a.ID = \mathcal{R}_2[j].a.ID) = \frac{1}{max\{|\mathit{Values}(\mathcal{R}_1, a.ID)|, |\mathit{Values}(\mathcal{R}_2, a.ID)|\}} \tag{5.11}$$

Hence, $joinCardinality(n, \theta_\tau^{(n)})$ for an edge join node n with $\mathfrak{t}_n =$
$\mathcal{R}_1 \bowtie_{a.ID=a.ID} \mathcal{R}_2$ and $children(n) = \{c_1, c_2\}$ is estimated according to Equation 5.12

$$\begin{aligned} joinCardinality(n, \theta_\tau^{(n)}) &= (|\mathcal{R}_1| * |\mathcal{R}_2|) * P(\mathcal{R}_1[i].a.ID = \mathcal{R}_2[j].a.ID) \\ &= \frac{joinCardinality(c_1, \theta_\tau^{(c_1)}) * joinCardinality(c_2, \theta_\tau^{(c_2)})}{max\{|\mathit{Values}(\mathcal{R}_1, a.ID)|, |\mathit{Values}(\mathcal{R}_2, a.ID)|\}} \end{aligned} \tag{5.12}$$

This formula poses the problem of determining $|\mathit{Values}(\mathcal{R}, a.ID)|$ for a relation $\mathcal{R}$ and a vertex variable a. The graph statistics provided by Gradoop include all the values needed here: the numbers of distinct source/target vertices overall as well as the numbers of distinct source/target vertices given a certain edge label. These values can be computed exactly, which might take some time for a large graph. Hence, in the temporal extension, they are only estimated. Section 6.2 explains this estimation in detail.

The Preservation of Values assumption introduced above justifies the uncomplicated subsequent use of this estimation formula. For example, suppose e is an edge variable with source vertex variable s and target vertex variable t. Suppose further, there is a query plan $\mathfrak{p}$ with root r

and $\mathfrak{t}_\mathfrak{p} = cand(s) \bowtie_{s.ID=s.ID} cand(e) \bowtie_{t.ID=t.ID} cand(t)$. Then, because of Preservation of Values, both join sequences $(cand(s) \bowtie_{s.ID=s.ID} cand(e)) \bowtie_{t.ID=t.ID} cand(t)$ and $cand(s) \bowtie_{s.ID=s.ID} (cand(e) \bowtie_{t.ID=t.ID} cand(t))$ will receive the same cardinality estimation.

These estimations can be used to calculate a non-recursive formula for estimating a query plan $\mathfrak{p}$'s *joinCardinality*. For every edge in the query pattern, there are two edge joins, as the edge has a source and target vertex. All other joins are cartesian products and value joins that can be handled like cartesian products here. As stated in Equation 5.7, filter and project nodes are irrelevant here. Hence,

$$joinCardinality(n, \theta_\tau^{(n)}) = \frac{\Pi_{m \text{ selection node } \in n} joinCardinality(m, \theta_\tau^{(m)})}{\Pi_{j \text{ edge join node } \in n, \text{joining edge e and vertex a}} max\{Values(cand(e), a.ID), Values(cand(a), a.ID)\}} \tag{5.13}$$

Plugging Equation 5.13 into Equation 5.4 concludes the discussion of query planning. The necessary extensions of this approach in order to process temporal GDL queries are simple.

5.1.5. From EPGM to TPGM Pattern Matching

The approach of Junghanns et al. discussed until now is not able to process temporal GDL queries (cf. Section 4) on TPGM graphs yet. However, it does not need to be modified crucially in order to handle such queries. In fact, the four special properties defined by TPGM, i.e. *tx-from*, *tx-to*, *val-from* and *val-to*, can be treated like any other property $k \in K$, at least in theory.

The next Section explains the implementation of the pattern matching component.

5.2. Implementation of (Temporal) Pattern Matching

Figure 5.3 summarizes the approach to pattern matching proposed by Junghanns et al. [28].

The following is devoted to explicate the implementation of the different aspects shown in Figure 5.3. First, the representation and post-processing of the query in Gradoop is explained in Section 5.2.1. After that, the implementation of graph relational tables will be outlined in Section 5.2.2. Section 5.2.3 then discusses the implementation of the query planning and execution components. Lastly, the final step in Figure 5.3, i.e. reconstructing the actual results, is briefly sketched in Section 5.2.4.

As the TPGM Pattern Matching operator is based on the EPGM implementation by Junghanns et al. [28], most aspects discussed in the following apply to both TPGM and EPGM matching implementations. Especially, the EPGM and thus large parts of the TPGM implementation presented here were realized by Junghanns et al [28]. Aspects exclusively relevant for temporal graphs and

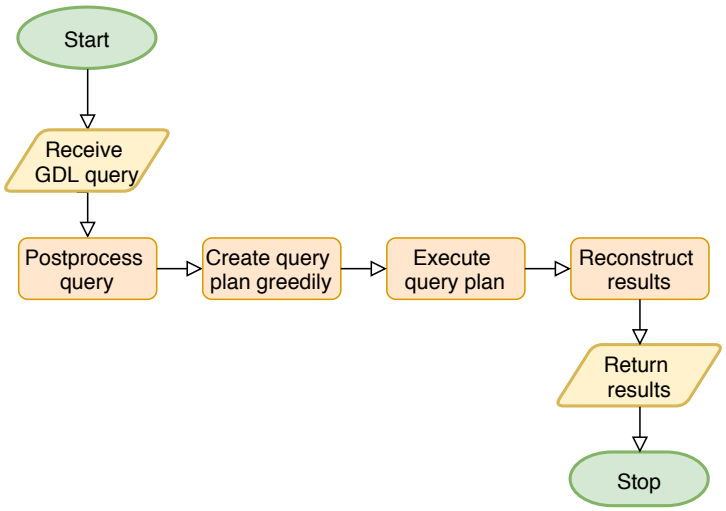

Figure 5.3.: Temporal Pattern Matching flow chart

queries are the only genuine contribution of this thesis w.r.t. to the Gradoop operator implementation. These aspects are marked explicitly throughout the discussion.

5.2.1. Query Handling

A GDL query is wrapped in Gradoop. For every data type in GDL, there is a corresponding wrapper type in Gradoop. As in this thesis some new data types were added to GDL, there were also new wrappers for them created in Gradoop.

The query constraints given in the GDL query are then converted to a conjunctive normal form (`CNF.java`) in Gradoop. Now, the representation of query constraints θ corresponds to Definition 9. A function is provided to obtain θ_X for any set of variables X.

In the temporal extension, the query must be implicitly augmented. For every query edge e connecting source vertex s to target vertex t, a predicate ensuring the overlap of e's, s's and t's valid intervals is added to the query CNF. The reason for this is that for the same graph element, different versions might exist in a database. These different versions have the same ID but may differ in other aspects, especially with regards to their transaction and valid intervals. Thus, it must be made sure that an edge matching e is actually valid at some point in time. Another special aspect of query handling in the temporal extension is a pipeline that transforms the CNF. It aims at improving the CNF so that it can be processed faster, of course without changing its semantics. This pipeline will be introduced in detail in Section 6.1, as it is a method to optimize the operator's performance.

5.2.2. Embeddings

Algorithm 2 is based on relational representations of vertices and edges. However, graph elements are stored as POJOs in Gradoop. It is thus necessary to convert these POJOs into a relational

representation in order to execute a query plan. Such a relational representation is called *embedding* in Junghanns et al. [28]'s implementation.

Embeddings are implemented as POJOs (`Embedding.java`) in Gradoop, their implementation is also described in Junghanns et al. [28]. Embeddings are optimized to store all necessary information in minimal space. In fact, an embedding contains only byte arrays. For every embedding, a metadata object is managed, that encapsulates the information on how to read the embedding's bytes. The reason for this focus on efficiency is that sets of embeddings represent the intermediate results, which are exchanged between workers during parallel processing. This exchange, and the application of Flink operators on the intermediate results (see Section 5.2.3) require a fast and compact data structure.

In detail, an embedding comprises three byte arrays:

- `idData` contains the IDs of the elements in the embedding. Every entry is actually a tuple (`flag`, `id`), where the byte `flag` distinguishes vertices and edges from paths.
- `idListData` is used to store paths. As path expressions are not relevant for this thesis, `idListData` is not discussed in detail here.
- `propData` stores all relevant property values

For the temporal extension of the matching operator, this implementation need not be changed. Temporal data is treated just like properties and hence stored in `propData`.

Note that the embedding implementation contains no information on which properties or time stamps belong to which graph element. Instead, this information is managed by a separate metadata object of type `EmbeddingMetaData.java`. Each embedding object is associated with such a metadata object. This object - also a POJO - stores mappings from query graph variables to the property and time values of their matching elements stored in the embedding. To give an example, Figure 5.4 shows an embedding and its metadata in a simplified way. The embedding contains data about three variables (s, e, t). It contains the ID value for every variable (42, 20, 08). Moreover the property *Name* is stored for s ("Bob") and the temporal property *tx-from* for e (1577836800000 ms since UNIX epoch, i.e. 2020-01-01T00:00:00 GMT). The meta data object contains three maps: `entryMapping` maps variables to the position of their IDs in `idData`. Similarly, `propertyMapping` maps properties of variables to the position of their values in `propData`. The map `directionMapping` is only relevant for path expressions and thus ignored here.

Embeddings and their metadata are now used to represent the (intermediate) results of a query execution. In what follows, it is shown how a query plan is implemented and executed on embeddings.

5.2.3. Query Plans and their Execution with Flink

An implementation of Greedy query plan construction as described in Section 5.1.2 is provided by Junghanns et al. [28]. The architecture is open to implementations of other query planning approaches.

Embedding

idData	[42, 20, 08]
idListData	
propData	["Bob", 1577836800000]

MetaData

entryMapping	s -> 0
	e -> 1
	t -> 2
directionMapping	
propertyMapping	(s, Name) -> 0
	(e, tx-from) -> 1

Figure 5.4.: Exemplary embedding and its metadata object (simplified)

A query plan is represented using different sub-classes of `PlanNode.java` in Gradoop. These classes, e.g. `FilterEmbeddingsNode.java`, correspond to the types of plan nodes defined in Section 5.1.2. Additionally, there is a class `ExpandEmbeddingsNode.java` that is responsible for path expressions and thus not treated here in detail.

All the `PlanNode.java` classes wrap an actual Flink operator, implemented as a subclass of the interface `PhysicalOperator.java`. This interface declares a method `evaluate()` that yields a Flink `DataSet` of `Embeddings`. `DataSets` in Flink simply contain a set of objects of the same type. Flink provides transformations (e.g. Filter, Map) on `DataSets` which are executed in parallel and yield another `DataSet`.

The `evaluate()` method in the subclasses of `PhysicalOperator.java`, and thus the actual query processing, is implemented using such transformations. For example, `FilterEmbeddingsNode.java` implements a filter node (Definition 25) and wraps the application of a `PhysicalOperator` named `FilterEmbeddings.java`. The Flink implementation of `FilterEmbeddings` applies a Flink `RichFilterFunction` transformation on the `DataSet`, yielding another `DataSet` of embeddings:

```
1   public class FilterEmbeddings implements PhysicalOperator {
2       //...
3       @Override
4       public DataSet<Embedding> evaluate() {
5           return input
6           .filter(new FilterEmbedding(predicates, metaData))
7           // ...
8       }
9       //...
10  }
11
12  public class FilterEmbedding extends RichFilterFunction<Embedding> {
13      // ...
14      private final CNF predicates;
15      // ...
16      private final EmbeddingMetaData metaData;
17      //...
18      @Override
19       public boolean filter(Embedding embedding) {
```

```
20              /**
21              * true iff predicates true on embedding
22              * described by metaData
23              */
24              return predicates.evaluate(embedding, metaData);
25          }
26      }
```

Listing 5.1: Snippet from Gradoop implementation of query plan execution

In the temporal extension, only the candidate selection for vertices and edges, done by `FilterAndProjectVertices.java` and `FilterAndProjectEdges.java`, must be implemented differently, as TPGM graph element POJOs store the interval data separately from normal properties. Hence, classes `FilterAndProjectTemporalVertices.java` and `FilterAndProjectTemporalEdges.java` replace `FilterAndProjectVertices.java` and `FilterAndProjectEdges.java` respectively. Besides that, the EPGM implementation is left unchanged.

If necessary, intermediate result embeddings are filtered for duplicate graph elements by the join operators. This is required when the matching strategy is set to isomorphism. In this case, as defined in Definition 12, two different query graph elements must not be mapped to the same data graph element.

Plan nodes are also responsible for creating the appropriate metadata object for the embeddings their wrapped operators yield.

The result of a query execution is a `DataSet` of embeddings, too. From this resulting `DataSet`, the returned `GraphCollection` is constructed.

5.2.4. Result Construction

It is clear, that each embedding encodes a match for the query, i.e. a graph. Hence, every match is described by and can be created from an embedding and its metadata object. The set of match graphs is a `GraphCollection.java`. An embedding of the final result stores the ID of every encoded element. The actual result is obtained by first creating in the usual Gradoop format for the encoded elements. These POJOs form `DataSet`s that are joined with the `DataSet`s of the database graph using their ID, which reconstructs all properties (including temporal properties in the TPGM extension). The reconstruction of properties is actually optional (see Appendix B)

The next Section introduces some aspects of the implementation that have been omitted so far. They are implemented as part of this thesis to optimize the run time.

6. Optimization

In order to speed up the implementation, two approaches are employed. Before executing the query, the temporal constraints in the query are rewritten. The rewriting methods are presented in detail in Section 6.1. Furthermore, the predicate selectivity estimations needed in Equation 5.4 were not implemented by Junghanns et al. [28]. In this thesis, an implementation is provided that is introduced in Section 6.2.

6.1. Query Rewriting

A query q consists of a graph pattern (V_q, E_q) and some constraints θ, as defined in Definition 9. Of course, changing the graph structure would change the semantics of the query. The constraints θ, however, are a boolean formula that could be reformulated in order to speed up query processing. Doing that, the semantics of θ must not be changed, evidently.

As already pointed out in Section 5.2.1, the query constraints are represented as conjunctive normal form (CNF) in Gradoop. Because of that, reformulating θ is not overly complicated. Rewriting θ here means to apply several transformations on the CNF, where each transformation yields a CNF again. Thus, query rewriting can be thought of as a pipeline that returns a new CNF. This returned CNF is semantically equivalent to the original one but Gradoop should be able to process it more quickly. Note that almost all transformation methods introduced here only refer to temporal constraints, as the thesis is focused on temporal queries and graphs.

There are two aspects of query processing that can be optimized by reformulating θ. First, θ might contain redundancies. Gradoop does not recognize redundant terms in a query, thus redundant terms are checked nevertheless. It is evident that checking terms costs time which could be saved by removing redundancies in θ. Second, θ might contain implicit information that could be used to filter out tuples earlier, reducing the size of intermediate embedding sets and thus the run time. Hence, these implicit information must be made explicit.

Additionally, a query involving temporal constraints can be checked for contradictions. If a contradiction is found, there is no need for query processing, as the result of a contradictory query is always empty.

In what follows, the transformations in the query rewriting pipeline are introduced. Each of the Sections 6.1.1-6.1.8 presents one transformation. An overview is provided in Section 6.1.9, followed by a few remarks on the implementation in Section 6.1.10.

Figure 6.1 shows the pipeline and also indicates which transformation is concerned with which of the aspects redundancy, implicit information and contradictions. Some transformations only serve as preparation for the following ones.

A CNF can be seen here as a set of sets of comparisons, every set of comparisons correspondig to a disjunctive clause. In the following, notations of CNFs as logical formulas and sets are used interchangeably, because sometimes the set notation is more convenient to express certain transformations, especially removal of clauses/comparisons.

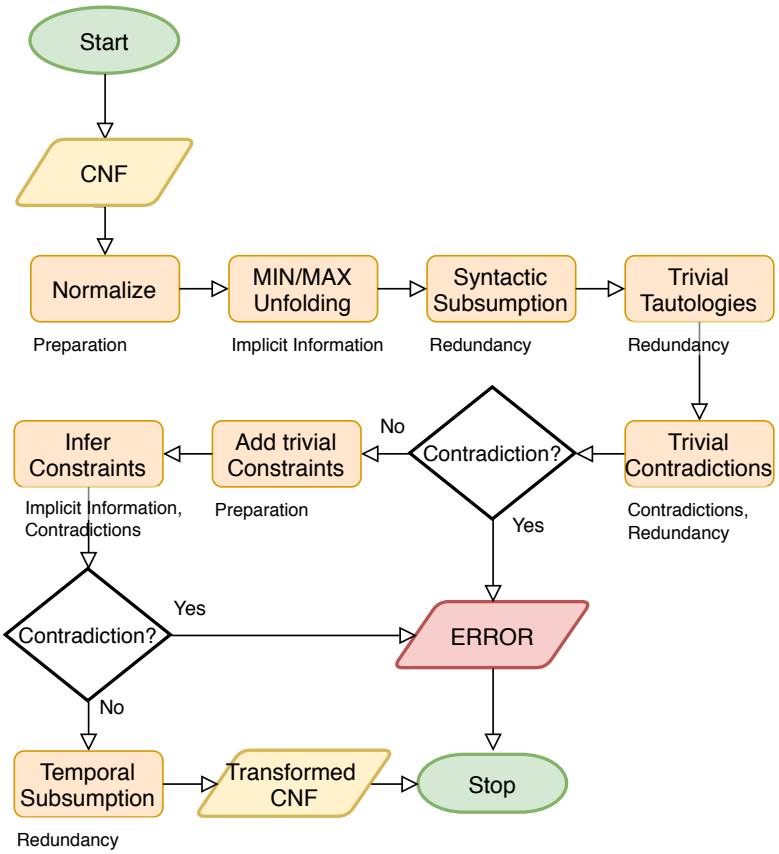

Figure 6.1.: Query transformation pipeline

The following GDL query serves as a running example throughout the discussion:

```
MATCH   (a)-[e]->(b)
WHERE    a.val.merge(Interval(b.val_from, Timestamp(2021-01-01))
            .longerThan(Minutes(1)) AND
         e.asOf(Timestamp(2020-06-01)) AND
         a.val_to >= e.val_to AND
         b.val_to >= e.val_to
```

It might look trivial at first, but its CNF is more complex. GDL adds a constraint $b.val\text{-}from \leq 2021\text{-}01\text{-}01$ to the query, because of the custom interval in the merge expression. Moreover, the merge expression leads to predicates ensuring overlap of $a.val$ and $[b.val\text{-}from,\ 2021\text{-}01\text{-}01]$ being added to the query ($max\{a.val\text{-}from,\ b.val\text{-}from\} \leq min\{a.val\text{-}to,\ 2021\text{-}01\text{-}01\}$). Gradoop then adds a constraint $max\{a.val\text{-}from,\ e.val\text{-}from,\ b.val\text{-}from\} \leq min\{a.val\text{-}to,\ e.val\text{-}to,\ b.val\text{-}to\}$ in order to force overlap of the edge and its two vertices. Thus, the CNF in question is given by

$$
\begin{gathered}
(Duration([max\{a.val\text{-}from,\ b.val\text{-}from\},\ min\{a.val\text{-}to,\ 2021\text{-}01\text{-}01\}]) > 60000ms) \\
\wedge\ (b.val\text{-}from \leq 2021\text{-}01\text{-}01]) \\
\wedge\ (max\{a.val\text{-}from,\ b.val\text{-}from\} \leq min\{a.val\text{-}to,\ 2021\text{-}01\text{-}01\}) \\
\wedge\ (e.val\text{-}from \leq 2020\text{-}06\text{-}01) \wedge (e.val\text{-}to \geq 2020\text{-}06\text{-}01) \\
\wedge\ (a.val\text{-}to \geq e.val\text{-}to)
\end{gathered}
$$

$$\wedge\ (b.\text{val-to} \geq e.\text{val-to})$$
$$\wedge\ (max\{a.\text{val-from}, e.\text{val-from}, b.\text{val-from}\} \leq min\{a.\text{val-to}, e.\text{val-to}, b.\text{val-to}\})$$

Before it is further processed, the CNF is normalized.

6.1.1. Comparison Normalization

A CNF consists of atoms that are comparisons of the form *lhs comp rhs*, where *lhs* and *rhs* are comparable elements like property selectors, time stamps etc. and $comp \in \{<, \leq, \neq, =, >, \geq\}$. So, there are overall six 'comparators'. Most transformations need to distinguish between comparators. To ease their implementation, the set of comparators in the CNF is reduced to $\{<, <=, \neq, =\}$. This is achieved by simply reformulating comparisons $a > b$ to $b < a$ and comparisons $a \geq b$ to $b \leq a$.

Algorithm 6 sums up this transformation.

Algorithm 6 Normalize

```
1:  procedure NORMALIZE(cnf)
2:      cnf' = empty CNF
3:      for clause ∈ cnf do
4:          clause' = empty disjunctive clause
5:          for comparison =: (lhs c rhs) ∈ clause do
6:              comparison' = comparison
7:              if c is > then
8:                  comparison' = rhs < lhs
9:              else if c is ≥ then
10:                 comparison' = rhs ≤ lhs
11:             clause' = clause' ∨ comparison'
12:         cnf' = cnf' ∧ clause'
13:     return cnf'
```

It is obvious that Algorithm 6 does not modify θ's semantics and yields a CNF.

Lemma 3. *For a CNF ϕ and a formula $\phi' :=$NORMALIZE(ϕ), ϕ' is a CNF and $\phi' \equiv \phi$.*

The application of normalization to the example CNF results in

$$(60000ms < (Duration([max\{a.\text{val-from}, b.\text{val-from}\}, min\{a.\text{val-to}, 2021\text{-}01\text{-}01\}])$$
$$\wedge\ (b.\text{val-from} \leq 2021\text{-}01\text{-}01])$$
$$\wedge\ (max\{a.\text{val-from}, b.\text{val-from}\} \leq min\{a.\text{val-to}, 2021\text{-}01\text{-}01\})$$
$$\wedge\ (e.\text{val-from} \leq 2020\text{-}06\text{-}01) \wedge (2020\text{-}06\text{-}01 \leq e.\text{val-to})$$
$$\wedge\ (e.\text{val-to} \leq a.\text{val-to})$$
$$\wedge\ (e.\text{val-to} \leq b.\text{val-to})$$
$$\wedge\ (max\{a.\text{val-from}, e.\text{val-from}, b.\text{val-from}\} \leq min\{a.\text{val-to}, e.\text{val-to}, b.\text{val-to}\})$$

Here, the comparisons switched by normalization are colored blue. Having normalized the CNF, transformations that actually improve query processing can be applied to it.

6.1.2. Min/Max Unfolding

In the extended GDL presented in Section 4, it is possible to compare several types of time stamps. Section 4.1.4 introduces two types of complex time stamps, MIN and MAX expressions. Both expressions take a set of simple time stamps as arguments, e.g. `MIN(a.tx-to, 2020-01-01, b.tx-from)` is a valid MIN expression. However, MIN and MAX expressions can not be nested, e.g. `MAX(a.tx-to, MIN(2020-01-01, b.tx-from))` is not allowed in the GDL extension.

In query processing, a potential problem concerning MIN and MAX expressions arises. A MIN/-MAX expression may contain more than one variable. Let M be a MIN/MAX expression and $a_1, ..., a_k$ be the variables contained in M. Then, every comparison c involving M can only be checked in a filter node n with $\{a_1, ..., a_k\} \subseteq var(n)$ by Algorithm 2. Stated differently, the check of c occurs rather late during query processing. As constraint checking typically filters out tuples and thus reduces the intermediate result cardinality, a late check may not be optimal. The earlier a constraint check is applied the smaller are the intermediate relations.

Comparisons involving MIN/MAX expressions can be seen as syntactic sugar for formulas over simple comparisons. These (sub-)formulas may be checked earlier than the whole MIN/MAX comparison, as they may include less variables. Lemma 4 states such formulas over simple comparisons that are equivalent to MIN/MAX comparisons.

Lemma 4. *Let $t_1, ..., t_k$ and x be time stamps. Then, the following equivalencies hold:*

$$(min\{t_1, ..., t_k\} = x) \quad \Leftrightarrow \quad \bigvee_{1 \leq i \leq k} (t_i = x \wedge (\bigwedge_{1 \leq j \leq k, j \neq i} t_i \leq t_j)) \tag{6.1}$$

$$(min\{t_1, ..., t_k\} \neq x) \quad \Leftrightarrow \quad \bigwedge_{1 \leq i \leq k} (t_i \neq x \vee (\bigvee_{1 \leq j \leq k, j \neq i} t_j < t_i)) \tag{6.2}$$

$$(min\{t_1, ..., t_k\} < x) \quad \Leftrightarrow \quad \bigvee_{1 \leq i \leq k} t_i < x \tag{6.3}$$

$$(min\{t_1, ..., t_k\} \leq x) \quad \Leftrightarrow \quad \bigvee_{1 \leq i \leq k} t_i \leq x \tag{6.4}$$

$$(min\{t_1, ..., t_k\} > x) \quad \Leftrightarrow \quad \bigwedge_{1 \leq i \leq k} t_i > x \tag{6.5}$$

$$(min\{t_1, ..., t_k\} \geq x) \quad \Leftrightarrow \quad \bigwedge_{1 \leq i \leq k} t_i \geq x \tag{6.6}$$

$$(max\{t_1, ..., t_k\} = x) \quad \Leftrightarrow \quad \bigvee_{1 \leq i \leq k} (t_i = x \wedge (\bigwedge_{1 \leq j \leq k, j \neq i} t_i \geq t_j)) \tag{6.7}$$

$$(max\{t_1, ..., t_k\} \neq x) \quad \Leftrightarrow \quad \bigwedge_{1 \leq i \leq k} (t_i \neq x \vee (\bigvee_{1 \leq j \leq k, j \neq i} t_j > t_i)) \tag{6.8}$$

$$(max\{t_1, ..., t_k\} < x) \quad \Leftrightarrow \quad \bigwedge_{1 \leq i \leq k} t_i < x \tag{6.9}$$

$$(max\{t_1, ..., t_k\} \leq x) \quad \Leftrightarrow \quad \bigwedge_{1 \leq i \leq k} t_i \leq x \tag{6.10}$$

$$(max\{t_1, ..., t_k\} > x) \quad \Leftrightarrow \quad \bigvee_{1 \leq i \leq k} t_i > x \tag{6.11}$$

$$(max\{t_1, ..., t_k\} \geq x) \quad \Leftrightarrow \quad \bigvee_{1 \leq i \leq k} t_i \geq x \tag{6.12}$$

Proof. Omitted, as the equations are trivial. □

These equations can now be used to translate comparisons involving MIN/MAX expressions into boolean formulas over simple comparisons between timestamps. This is also true for comparisons between two MIN/MAX expressions. E.g.,

$$\begin{aligned} min\{t_1, t_2\} < max\{t_3, t_4\} \quad &\Leftrightarrow \quad t_1 < max\{t_3, t_4\} \vee t_2 < max\{t_3, t_4\} \\ &\Leftrightarrow \quad (t_1 < t_3) \vee (t_1 < t_4) \vee (t_2 < t_3) \vee (t_2 < t_4) \end{aligned}$$

by equations 6.3 and 6.12.

Not all of the provided equations are actually applied to θ in the transformation. The (in)equality equations (Equations 6.1, 6.2, 6.7, 6.8) yield quite complex formulas containing k^2 simple comparisons. Because of this complexity, the Equations 6.1, 6.2, 6.7 and 6.8 are not applied.

Some equations (Equations 6.5, 6.6, 6.9, 6.10), however, yield a conjunction of k comparisons, i.e. a CNF consisting of k singleton clauses. They are applied in the transformation, but only to comparisons that constitute a singleton clause in the input CNF. The result of this application is then immediately a CNF again. Here, a comparison c involving variables $a_1, ..., a_l$ was reduced to singleton CNF clauses, each involving at most two variables. These clauses can now be, in general, checked earlier than c and might thus contribute to reducing the size of intermediate embedding sets during query processing. Another advantage is that simple comparisons can be handled more easily by subsequent transformations.

Equations yielding a disjunction (i.e. Equations 6.3, 6.4, 6.11, 6.12) are only applied to comparisons within a non-singleton CNF clause. Evidently, this clause is then a non-singleton CNF clause again, so the CNF structure is preserved. These reformulations do not immediately help in query processing, because the resulting clause still contains all the variables $t_1, ..., t_k$. The reason to apply them is that simple comparisons simplify the application of further transformations later and may also be simplified by these transformations.

The approach is summarized in Algorithm 7.

Algorithm 7 Unfold MIN/MAX

```
1:  procedure UnfoldMinMax(cnf)
2:      cnf' = empty CNF
3:      changed = TRUE
4:      while changed do
5:          changed = FALSE
6:          cnf' = empty CNF
7:          for clause ∈ cnf do
8:              clause' = empty disjunctive clause
9:              for comparison ∈ clause do
10:                 comparison' = comparison
11:                 if Equation 6.3, 6.4, 6.5, 6.6, 6.9, 6.10, 6.11 or 6.12 applicable without changing
    CNF structure then
12:                     comparison' = apply an equation to comparison
13:                     changed = TRUE
14:                 clause' = clause' ∨ comparison'
15:             cnf' = cnf' ∧ clause'
16:     return cnf'
```

As pointed out above, Algorithm 7. yields a CNF with the same semantics as the input CNF.

Lemma 5. *For a CNF ϕ and a formula $\phi' :=$ unfoldMinMax(ϕ), ϕ' is a CNF and $\phi' \equiv \phi$.*

Proof. The equivalence $\phi' \equiv \phi$ is evident from the equations given in Lemma 4.

In all three cases, the result is again a CNF. The algorithm terminates, because after a finite number of steps, none of the equations is applicable anymore, causing *changed* to be FALSE in line 4. □

The application of Algorithm 7 to the example query results in

$$
\begin{gathered}
\textit{(60000ms} < \textit{(Duration([max\{a.val-from, b.val-from\}, min\{a.val-to, 2021-01-01\}])} \\
\wedge \textit{(a.val-from} \leq \textit{a.val-to)} \wedge \textit{(a.val-from} \leq \textit{2021-01-01)} \wedge \textit{(b.val-from} \leq \textit{a.val-to)} \wedge \textit{(b.val-from} \leq \\
\textit{2021-01-01)} \\
\wedge \textit{(e.val-from} \leq \textit{2020-06-01)} \wedge \textit{(2020-06-01} \leq \textit{e.val-to)} \\
\wedge \textit{(e.val-to} \leq \textit{a.val-to)} \\
\wedge \textit{(e.val-to}\leq \textit{b.val-to)} \\
\wedge \textit{(a.val-from} \leq \textit{a.val-to)} \wedge \textit{(a.val-from} \leq \textit{e.val-to)} \wedge \textit{(a.val-from} \leq \textit{b.val-to)} \wedge \textit{(e.val-from} \leq \\
\textit{a.val-to)} \wedge \textit{(e.val-from} \leq \textit{e.val-to)} \wedge \textit{(e.val-from} \leq \textit{b.val-to)} \wedge \textit{(b.val-from} \leq \textit{a.val-to)} \wedge \\
\textit{(b.val-from} \leq \textit{e.val-to)} \wedge \textit{(b.val-from} \leq \textit{b.val-to)}
\end{gathered}
$$

Clauses replacing MIN/MAX clauses are colored blue.

The application of unfoldMinMax might increase the CNF's redundancy since clauses added by the algorithm might already be present in the CNF. The next two transformations in the pipeline aim at reducing redundancy.

6.1.3. Syntactic Subsumption

Subsumption is a well-known operation on CNFs. A CNF clause c_1 is said to *subsume* another clause c_2 if $c1 \Rightarrow c2$. If a clause is subsumed by another clause in the CNF, it is redundant and can thus be removed from the CNF without changing the semantics.

In propositional logic, atoms are simply variables, which may be negated. To give an example, $\psi = (A \vee B) \wedge (B \vee \neg C) \wedge (A)$ is a CNF. Every clause c_i in such a CNF can thus be represented as a set of (potentially negated) variables, e.g. for ψ these sets would be $c_1 = \{A, B\}$, $c_2 = \{B, \neg C\}$ and $c_3 = \{A\}$. Given this representation, a clause c_i evidently subsumes another clause c_j $(i \neq j)$ if and only if $c_i \subseteq c_j$. In ψ, $c_3 \subseteq c_1$. Consequently, ψ can be simplified to $(B \vee \neg C) \wedge (A)$.

The syntactic subsumption transformation executes such subsumptions for the input CNF. Its normalized comparisons can be interpreted as propositional atoms by reducing them to their string representation. For example, a comparison *a.tx-to<2020-01-01* is now simply considered an atom called '*a.tx-to<2020-01-01*'.

This allows to treat the CNF as a propositional logic formula, so that subsumption can be applied to it. The subsumption here is purely syntactical, hence the term 'syntactic subsumption'. Technically spoken, the string representations of comparisons are checked for equality, i.e. a comparison atom a_1 implies another comparison atom a_2, iff *string*$(a_1) =$ *string*(a_2). More sublime implications like $(b.tx\text{-}to < 2018\text{-}01\text{-}01) \Rightarrow (b.tx\text{-}to < 2020\text{-}01\text{-}01)$ are not detected here. However, such implications will be dealt with later, see Section 6.1.7.

In Algorithm 8, syntactic subsumption is formalized.

Algorithm 8 Syntactic Subsumption

```
1:  procedure SyntacticSubsumption(cnf)
2:      cnf' = array of clauses in cnf sorted by their size
3:      cnf'' = empty CNF
4:      subsumed[] = new boolean array, initialized with FALSE
5:      for all i (1 ≤ i ≤ |cnf|) do
6:          if not subsumed[i] then
7:              for all j (i < j ≤ |cnf|) do
8:                  if not subsumed[j] then
9:                      if all comparisons in cnf'[i] also occur in cnf'[j] then
10:                         subsumed[j] = TRUE
11:             cnf'' = cnf'' ∧ cnf'[i]
12:     return cnf''
```

Lemma 6. *For a CNF ϕ and a formula ϕ' := syntacticSubsumption(ϕ), ϕ' is a CNF and $\phi' \equiv \phi$.*

Proof. As subsumption only removes disjunctive clauses from the input CNF, its result is a CNF, too. Subsumption obviously preserves the semantics of the CNF ϕ, thus $\phi' \equiv \phi$. □

As an example for syntactic subsumption, consider its effect on the running example CNF:

$$(60000ms < (Duration([max\{a.val\text{-}from,\ b.val\text{-}from\},\ min\{a.val\text{-}to,\ 2021\text{-}01\text{-}01\}])$$
$$\wedge\ (a.val\text{-}from \leq a.val\text{-}to) \wedge (a.val\text{-}from \leq 2021\text{-}01\text{-}01) \wedge (b.val\text{-}from \leq a.val\text{-}to) \wedge (b.val\text{-}from \leq 2021\text{-}01\text{-}01)$$
$$\wedge\ (e.val\text{-}from \leq 2020\text{-}06\text{-}01) \wedge (2020\text{-}06\text{-}01 \leq e.val\text{-}to)$$
$$\wedge\ (e.val\text{-}to \leq a.val\text{-}to)$$
$$\wedge\ (e.val\text{-}to \leq b.val\text{-}to)$$
$$\sout{\wedge\ (a.val\text{-}from \leq a.val\text{-}to)} \wedge (a.val\text{-}from \leq e.val\text{-}to) \wedge (a.val\text{-}from \leq b.val\text{-}to) \wedge (e.val\text{-}from \leq a.val\text{-}to) \wedge (e.val\text{-}from \leq e.val\text{-}to) \wedge (e.val\text{-}from \leq b.val\text{-}to) \sout{\wedge\ (b.val\text{-}from \leq a.val\text{-}to)} \wedge (b.val\text{-}from \leq e.val\text{-}to) \wedge (b.val\text{-}from \leq b.val\text{-}to)$$

Clauses subsumed by another clause are striked through.

In the next transformation, the CNF is further simplified.

6.1.4. Trivial Tautologies

A query CNF may contain comparisons that are always true by definition. If such a *trivial tautology* occurs within a clause, this clause always evaluates to true and can thus be removed from the CNF.

The trivial tautologies transformation detects three types of tautologies:

- Comparisons $x = x$ and $x \leq x$ are obviously tautological for every numerical or temporal comparable x
- For a query variable a, comparisons of the form $a.tx\text{-}from \leq a.tx\text{-}to$ and $a.val\text{-}from \leq a.val\text{-}to$ always hold by the definition of transaction and valid interval
- If two literals l_1 and l_2 are compared, it can be immediately checked whether the comparison holds or not. In case it holds, a tautology was detected. Otherwise, the comparison is contradictory (see Section 6.1.5)

Typically, the user will not state such trivial tautologies deliberately. They may, however, be implicitly created by GDL and transformations within Gradoop. E.g., creating an interval `Interval(MIN(a.tx_from, Timestamp(2018-01-01)), MAX(a.tx_to Timestamp(2019-01-01)))` in a GDL query would add the constraint $max\{a.tx\text{-}from, 2018\text{-}01\text{-}01\} \leq min\{a.tx\text{-}to, 2019\text{-}01\text{-}01\}$ to the query. The MIN/MAX unfolding transformation (Section 6.1.2) would then reformulate this to $(a.tx\text{-}from \leq a.tx\text{-}to) \wedge (a.tx\text{-}from \leq 2019\text{-}01\text{-}01) \wedge (2018\text{-}01\text{-}01 \leq a.tx\text{-}to) \wedge (2018\text{-}01\text{-}01 \leq 2019\text{-}01\text{-}01)$. Applying trivial tautologies to this expression simplifies it to $(a.tx\text{-}from \leq 2019\text{-}01\text{-}01) \wedge (2018\text{-}01\text{-}01 \leq a.tx\text{-}to)$.

Algorithm 9 summarises the approach to removing trivial tautologies.

Algorithm 9 Trivial Tautologies

```
1: procedure TrivialTautologies(cnf)
2:     for all clause ∈ cnf do
3:         for all comparison ∈ clause do
4:             if comparison is a trivial tautology then
5:                 cnf = cnf \ {clause}
6:                 continue with next clause
7:     return cnf
```

Lemma 7. *For a CNF ϕ and a formula $\phi' :=$ trivialTautologies(ϕ), ϕ' is a CNF and $\phi' \equiv \phi$.*

Proof. Algorithm 9 modifies ϕ only by removing clauses. Hence, ϕ' is a CNF. Furthermore, it is clear that $\phi' \equiv \phi$, as only tautological clauses are removed. □

When applied to the running example, the trivial tautologies transformation yields

$$(60000ms < (Duration([max\{a.val\text{-}from,\ b.val\text{-}from\},\ min\{a.val\text{-}to,\ 2021\text{-}01\text{-}01\}])$$
$$\wedge \sout{(a.val\text{-}from \leq a.val\text{-}to)} \wedge (a.val\text{-}from \leq 2021\text{-}01\text{-}01) \wedge (b.val\text{-}from \leq a.val\text{-}to) \wedge (b.val\text{-}from \leq 2021\text{-}01\text{-}01)$$
$$\wedge (e.val\text{-}from \leq 2020\text{-}06\text{-}01) \wedge (2020\text{-}06\text{-}01 \leq e.val\text{-}to)$$
$$\wedge (e.val\text{-}to \leq a.val\text{-}to)$$
$$\wedge (e.val\text{-}to \leq b.val\text{-}to)$$
$$\wedge (a.val\text{-}from \leq e.val\text{-}to) \wedge (a.val\text{-}from \leq b.val\text{-}to) \wedge (e.val\text{-}from \leq a.val\text{-}to) \sout{\wedge (e.val\text{-}from \leq e.val\text{-}to)} \wedge (e.val\text{-}from \leq b.val\text{-}to)$$
$$\wedge (b.val\text{-}from \leq e.val\text{-}to) \sout{\wedge (b.val\text{-}from \leq b.val\text{-}to)}$$

Again, clauses removed by the transformation are striked through.

To every tautological comparison c corresponds the trivial contradiction given by $\neg c$. These contradictions can sometimes also be used to simplify the query.

6.1.5. Trivial Contradictions

If a disjunctive clause in the CNF contains a contradictory comparison, this comparison can be removed from the clause. In case the contradictory comparison makes up a singleton clause, the whole query is contradictory. Then, an empty set can be immediately returned as the query's result.

The trivial contradictions transformation checks the input CNF for contradictions that are equal to the trivial tautologies' (Section 6.1.4) negations:

- Comparisons $x \neq x$ and $x < x$ are contradictory for every numerical or temporal comparable x. The same holds for comparisons $x > x$. However, the query does not contain such comparisons anymore, as they are removed by normalization (Section 6.1.1).

- For a query variable a, comparisons of the form $a.tx\text{-}to < a.tx\text{-}from$ and $a.val\text{-}to < a.val\text{-}from$ violate the definition of transaction and valid intervals and are thus contradictory. Comparisons like $a.tx\text{-}to \leq a.tx\text{-}from$ are not contradictory, as $a.tx\text{-}to = a.tx\text{-}from$ could be possible.
- If two literals l_1 and l_2 are compared, it can be immediately checked whether the comparison holds or not

As for trivial tautologies, it can be assumed that the user does not state trivial contradictions on purpose. Just like trivial tautologies, they may be created by implicit manipulation of the issued query during rewriting pipeline.

The algorithm for the trivial contradiction transformation is given by Algorithm 10. Other than the transformations before, it may terminate the whole pipeline by returning an error for a contradictory query (line 6).

Algorithm 10 Trivial Contradictions

```
1:  procedure TrivialContradictions(cnf)
2:      for all clause ∈ cnf do
3:          for all comparison ∈ clause do
4:              if comparison is a trivial contradiction then
5:                  if |clause| = 1 then
6:                      query contradictory, throw error
7:                  else
8:                      clause = clause \ {comparison}
9:                      continue with next comparison
10:     return cnf
```

Lemma 8. *For a CNF ϕ,* trivialContradictions(ϕ) *returns either a CNF ϕ' or an error. If an error is returned, ϕ is contradictory. Otherwise, $\phi' \equiv \phi$.*

Proof. In case trivialContradictions does not return an error, the returned formula ϕ' is clearly a CNF. Algorithm 10 only removes comparisons within clauses from the input CNF ϕ, thus the CNF structure is never violated. Furthermore, $\phi' \equiv \phi$, because only contradictions are removed, which are not satisfiable in any case.

An error is returned only if a single, thus necessary, clause in ϕ states one of the obvious contradictions listed above or all comparisons within a clause are contradictory. Because of that, an error always implies a contradictory query. □

As the running example does not contain trivial contradictions, the execution of Algorithm 10 does not change it and thus results in

$$\begin{aligned}&(60000ms < (Duration([max\{a.val\text{-}from,\ b.val\text{-}from\},\ min\{a.val\text{-}to,\ 2021\text{-}01\text{-}01\}])\\ &\wedge\ (a.val\text{-}from \leq 2021\text{-}01\text{-}01) \wedge (b.val\text{-}from \leq a.val\text{-}to) \wedge (b.val\text{-}from \leq 2021\text{-}01\text{-}01)\\ &\wedge\ (e.val\text{-}from \leq 2020\text{-}06\text{-}01) \wedge (2020\text{-}06\text{-}01 \leq e.val\text{-}to)\\ &\wedge\ (e.val\text{-}to \leq a.val\text{-}to)\end{aligned}$$

$$\wedge \textit{ (e.val-to} \leq \textit{ b.val-to)}$$
$$\wedge \textit{ (a.val-from} \leq \textit{ e.val-to)} \wedge \textit{ (a.val-from} \leq \textit{ b.val-to)} \wedge \textit{ (e.val-from} \leq \textit{ a.val-to)}$$

$$\wedge \textit{ (e.val-from} \leq \textit{ b.val-to)}$$

$$\wedge \textit{ (b.val-from} \leq \textit{ e.val-to)}$$

Of course, the absence of an error in Algorithm 10 does not necessarily imply satisfiability. There can be more subtle contradictions that can not be described on the level of just one contradiction. For example, consider the CNF $\textit{a.tx-to} < \textit{b.tx-from} \wedge \textit{b.tx-from} < \textit{2020-01-01} \wedge \textit{2020-03-01} < \textit{a.tx-from}$. Here, the first two constraints imply $\textit{a.tx-to} < \textit{2020-01-01}$, which, together with the third constraint, implies $\textit{a.tx-to} < \textit{a.tx-from}$. Clearly, Algorithm 10 would not notice this contradiction. The two transformations introduced next, however, are capable of detecting such non-trivial contradictions.

6.1.6. Adding Trivial Constraints

This transformation only prepares the input CNF for the following transformation, Bounds Inference (Section 6.1.7). It adds redundant temporal information to the query. The intuition is that the query should now explicitly express everything that is known about the relations between involved time stamps. However, the transformation only considers time stamps in singleton clauses and no MIN/MAX expressions. These clauses encode necessary constraints that must always hold in order to satisfy the CNF.

Two types of constraints are added to the CNF as singleton clauses. For every query variable a, the comparison $\textit{a.tx-from} \leq \textit{a.tx-to}$ is added, provided that both properties are referenced in singleton clauses somewhere. The same is done for the valid times. Furthermore, for every pair of time stamps (t_1, t_2) with $(t_1 \neq t_2)$, a $<$ comparison describing their relation is added. Again, only time stamps occurring in singleton clauses are considered. The resulting CNF is obviously normalized again, i.e. only the comparators $\leq, <, =$ and $\neq$ are used.

Algorithm 11 sums up the transformation.

Algorithm 11 Add Trivial Constraints

```
1:  procedure AddTrivialConstraints(cnf)
2:      cnf_singletons = {C_i ∈ cnf | |C_i| = 1, no MIN/MAX contained in C_i}
3:      cnf_add = empty CNF
4:      for all var, var variable referenced in cnf_singletons do
5:          if var.tx-from and var.tx-to referenced in cnf_singletons then
6:              cnf_add = cnf_add ∧ (var.tx-from ≤ var.tx-to)
7:          if var.val-from and var.val-to referenced in cnf_singletons then
8:              cnf_add = cnf_add ∧ (var.val-from ≤ var.val-to)
9:      T = all time literals referenced in cnf_singletons
10:     for t_i, t_j ∈ T, t_i ≠ t_j do
11:         if t_i < t_j then
12:             cnf_add = cnf_add ∧ (t_i < t_j)
13:         else
14:             cnf_add = cnf_add ∧ (t_j < t_i)
15:     return cnf ∧ cnf_add
```

Lemma 9. *For a CNF ϕ and a formula ϕ' :=*addTrivialConstraints*(ϕ), ϕ' is a CNF and $\phi' \equiv \phi$.*

Proof. As Algorithm 11 only modifies ϕ by adding singleton clauses, ϕ' is clearly a CNF. The semantics of ϕ are not changed as all added clauses express trivial tautologies, hence $\phi' \equiv \phi$. □

The application of Algorithm 11 to the running example results in

$$
\begin{gathered}
\textit{(60000ms} < \textit{(Duration([max\{a.val-from, b.val-from\}, min\{a.val-to, 2021-01-01\}])} \\
\wedge \textit{(a.val-from} \leq \textit{2021-01-01)} \wedge \textit{(b.val-from} \leq \textit{a.val-to)} \wedge \textit{(b.val-from} \leq \textit{2021-01-01)} \\
\wedge \textit{(e.val-from} \leq \textit{2020-06-01)} \wedge \textit{(2020-06-01} \leq \textit{e.val-to)} \\
\wedge \textit{(e.val-to} \leq \textit{a.val-to)} \\
\wedge \textit{(e.val-to} \leq \textit{b.val-to)} \\
\wedge \textit{(a.val-from} \leq \textit{e.val-to)} \wedge \textit{(a.val-from} \leq \textit{b.val-to)} \wedge \textit{(e.val-from} \leq \textit{a.val-to)} \wedge \textit{(e.val-from} \leq \\
\textit{b.val-to)} \\
\wedge \textit{(b.val-from} \leq \textit{e.val-to)} \\
\wedge \textit{(a.val-from} \leq \textit{a.val-to)} \wedge \textit{(e.val-from} \leq \textit{e.val-to)} \wedge \textit{(b.val-from} \leq \textit{b.val-to)} \wedge \textit{(2020-06-01} \leq \\
\textit{2021-01-01)}
\end{gathered}
$$

Green clauses are those added by Algorithm 11.

The transformation raises the question why the Trivial Tautologies transformation (Section 6.1.4) has been applied before. After all, this transformation removed tautologies that are now added again. The reason for the earlier application of Trivial Tautologies is that it may also remove disjunctive, i.e. non-singleton clauses, thus reducing the query's complexity.

Unlike transformations like MIN/MAX unfolding, it would not be reasonable to apply only Algorithm 11. Its sole purpose is to provide the input for the Infer Bounds transformation (Section

6.1.7). In fact, addTrivialConstraints harms the query by adding constraints without adding actual information. Note that the trivial tautologies added here will be removed again later. Before this can be done, they are, however, crucial for the next transformation in the pipeline.

6.1.7. Inferring helpful Constraints

A CNF may include implicit information about time stamps. The transformation introduced here seeks to make such implicit constraints explicit, as this might improve the run time. As an example, consider the CNF ($a.tx\text{-}from < b.tx\text{-}from \wedge b.tx\text{-}to < 2019\text{-}01\text{-}01$). Clearly, as $b.tx\text{-}from \leq b.tx\text{-}to$ it can be concluded that ($a.tx\text{-}from < 2019\text{-}01\text{-}01$). Obviously, this implicit constraint adds some information about a. Thus, adding it to the CNF will probably lead to less tuples for a passing the candidate selection step of query processing (cf. Algorithm 2). In other words $\theta_{\{a\}}$ is more specific now. As the size of intermediate results impacts the run time, it makes sense to add the clause to the CNF.

The transformation introduced here only considers clauses that are bot temporal and necessary, i.e. singleton clauses, without possibly remaining MIN/MAX expressions. These clauses are denoted *clauses* in the following.

Implicit constraints are made explicit via transitive closures. Because of normalization, all comparisons to consider here are of the form $a\ comp\ b$ where $comp \in \{<, \leq, =, \neq\}$.

First, all comparisons of type $=$, $\leq$ or $<$ implied by *clauses* are determined. More formally, relations $R_{comp} = \{(a, b) \mid clauses \Rightarrow (a\ comp\ b)\}$ are sought for $comp \in \{<, \leq, =\}$. Furthermore, it should hold that $(a, b) \in R_{=} \Rightarrow (a, b) \notin R_{\leq}$ and $(a, b) \in R_{<} \Rightarrow (a, b) \notin R_{\leq}$ in order to prohibit redundancy. To compute these relations, relations $C_{comp} := \{(a, b) | (a\ comp\ b) \in clauses\}$ are created for $comp \in \{\leq, <, =\}$. With these relations the desired relations R_{comp} can be computed using transitive closures ($\circ$ denotes the composition of two relations):

$$R_{=} = C_{=}^{+} \cup \{(a, b) \in (C_{=} \cup C_{\leq})^{+} \mid (b, a) \in (C_{=} \cup C_{\leq})^{+}\} \tag{6.13}$$

$$R_{\leq} = (R_{=} \cup C_{\leq})^{+} \setminus R_{=} \tag{6.14}$$

$$R_{<} = (((R_{=} \cup R_{\leq}) \circ C_{<}) \cup (C_{<} \circ (R_{=} \cup R_{\leq})) \cup C_{<})^{+} \tag{6.15}$$

The relations R_{comp} are now used to compute lower and upper bounds and to detect errors. Algorithm 12 is a high-level description of the method.

Algorithm 12 Infer Bounds

```
1: procedure InferBounds(cnf)
2:     clauses = singleton temporal clauses from cnf, not containing MIN or MAX
3:     lower = map, lower[s] = −∞ for every selector s referenced in clauses
4:     upper = map, upper[s] = ∞ for every selector s referenced in clauses
5:     Compute R_≤, R_< and R_= from clauses                    ▷ Eq. 6.13, 6.14, 6.15
6:     for all (x, y) ∈ R_< do
7:         if x == y then
8:             query contradictory, throw error
9:     for all (a, b) ∈ R_= do
10:        lower, upper = updateEQ(lower, upper, (a, b))
11:        if updateEQ threw error then
12:            query contradictory, throw error
13:    for all (a, b) ∈ R_≤ do
14:        lower, upper = updateLEQ(lower, upper, (a, b))
15:        if updateLEQ threw error then
16:            query contradictory, throw error
17:    for all (a, b) ∈ R_< do
18:        lower, upper = updateLT(lower, upper, (a, b))
19:        if updateLT threw error then
20:            query contradictory, throw error
21:    if NEQContradiction(lower, upper, clauses) then
22:        query contradictory, throw error
23:    clauses' = newClausesFromBounds(clauses, lower, upper)
24:    return (cnf\clauses) ∪ clauses'
```

Before any bounds are computed, the relation $R_<$ is checked for contradictions $x < x$ in lines 6-8. If such a contradiction is implied by the query, the whole query is contradictory and need not be processed further. Due to the construction of the relations R_{comp} and the adding of trivial constraints like $a.tx\text{-}from \leq a.tx\text{-}to$ before (Section 6.1.6), an implicit contradiction $a.tx\text{-}to < a.tx\text{-}from$ would lead to a clause $(a.tx\text{-}to, a.tx\text{-}to) \in R_<$. Hence, such contradictions are also detected in lines 6-8.

The following algorithms fill in the remaining details. Updating the lower and upper bound values is straightforward and the correctness of Algorithms 13 - 15 evident.

Algorithm 13 Update bounds with equality constraints

```
1:  procedure UpdateEQ(lower, upper, (a, b))
2:      if a is a literal and b a selector then
3:          selector = b
4:          literal = a
5:      else if a is a selector and b a literal then
6:          selector = a
7:          literal = b
8:      else
9:          return lower, upper
10:
11:     if lower[selector] > literal then
12:         query contradictory, throw error
13:     else if upper[selector] < literal then
14:         query contradictory, throw error
15:     else
16:         lower[selector] = literal
17:         upper[selector] = literal
18:     return lower, upper
```

Algorithm 14 Update bounds with $\leq$ constraints

```
1:  procedure UpdateLEQ(lower, upper, (a, b))
2:      if a is a literal and b a selector then
3:          lower[b] = max{lower[b], a}
4:          if lower[b] > upper[b] then
5:              query contradictory, throw error
6:      else if a is a selector and b a literal then
7:          upper[a] = min{upper[a], b}
8:          if lower[a] > upper[a] then
9:              query contradictory, throw error
10:     return lower, upper
```

Algorithm 15 Update bounds with $<$ constraints

```
1:  procedure UpdateLT(lower, upper, (a, b))
2:      if a is a literal and b a selector then
3:          lower[b] = max{lower[b], a + 1}
4:          if lower[b] > upper[b] then
5:              query contradictory, throw error
6:      else if a is a selector and b a literal then
7:          upper[a] = min{upper[a], b − 1}
8:          if lower[a] > upper[a] then
9:              query contradictory, throw error
10:     return lower, upper
```

By construction of these algorithms, the inferred bounds do not contradict the comparisons of type $<$, $\leq$ and $=$ in *clauses*. However, it could be possible that a comparison of type $\neq$ contradicts

a newly infered bound. The function NEQContradiction investigates such contradictions and throws an error if it detects one. It is given in Algorithm 16.

Algorithm 16 Check for NEQ contradictions

```
1: procedure NEQContradiction(lower, upper, clauses)
2:     for all (a ≠ b) ∈ clauses do
3:         if a selector and b selector then
4:             if lower[a] == upper[a] ∧ lower[b] == upper[b] ∧ lower[a] == lower[b] then
5:                 query contradictory, throw error
6:         else if a selector and b literal then
7:             if lower[a] == upper[a] ∧ lower[a] == b then
8:                 query contradictory, throw error
9:         else if a literal and b selector then
10:            if lower[b] == upper[b] ∧ lower[b] == a then
11:                query contradictory, throw error
```

After this contradiction check, new clauses are constructed from the inferred bounds. Algorithm 17 shows the function newClausesFromBounds called in line 23 in Algorithm 12.

Algorithm 17 construct new clauses from bounds

```
1: procedure newClausesFromBounds(lower, upper, clauses)
2:     clauses' = ∅
3:     for all clause ∈ clauses do
4:         if clause == (s1 comp s2), s1 and s2 selectors then
5:             if NOT (clause of the form a.int_from ≤ a.int_to (int ∈ {tx, val}) or l1 < l2 (l1,
   l2 literals) then
6:                 clauses' = clauses' ∧ clause
7:     for all s selector with bounds in lower and upper do
8:         if upper[s] == lower[s] then
9:             clauses' = clauses' ∧ (s = lower[s])
10:        if lower[s] > −∞ then
11:            if NOT (s of the form a.int−to (int ∈ {tx, val}) ∧ lower[a.int−from] == lower[s])
   then
12:                clauses' = clauses' ∧ (lower[s] ≤ s)
13:        if upper[s] < ∞ then
14:            if NOT (s of the form a.int−from (int ∈ {tx, val}) ∧ upper[a.int−to] == upper[s])
   then
15:                clauses' = clauses' ∧ (s ≤ upper[s])
16:    return clauses'
```

Line 5 deletes the trivial tautologies added before. The conditions in lines 10-14 ensure that no uninformative comparisons are added to the query. E.g., the lower bound of *a.tx-from* is evidently a trivial, but not informative lower bound for *a.tx-to* (line 11). Analogously, the upper bound of *a.tx-to* is a trivial upper bound of *a.tx-from* (line 14).

Lemma 10. *For a CNF ϕ,* inferBounds(ϕ) *returns either a CNF ϕ' or an error. If an error is returned, ϕ is contradictory. Otherwise, $\phi' \equiv \phi$.*

Proof. If no error is thrown, ϕ' is a CNF. Only the necessary, i.e. singleton, clauses of ϕ are manipulated by Algorithm 12 (line 24 of Algorithm 12). This manipulation is given by Algorithm 17. Here, a conjunction of single clauses and hence, a CNF is built in lines 6, 12 and 15. Because of that, the return value ϕ' of Algorithm 12 is again a CNF, if no error is thrown.

Algorithm 12 is correct, i.e. $\phi' \equiv \phi$ if no error is thrown. Moreover, errors are only thrown if ϕ is contradictory. The correctness of the bounds computations is obvious from Equations 6.13, 6.14 and 6.15 and the trivial Algorithms 13, 14 and 15. Errors are thrown if newly inferred bounds contradict the necessary clauses they are inferred from. For clauses with comparison type $<$, $\leq$ and $=$, this is done implicitly: as soon as any lower bound would be greater than the corresponding upper bound, an error is thrown in Algorithms 13, 14 and 15. For inequality comparisons, Algorithm 16 explicitly detects contradictions and throws errors if necessary. Thus, errors are only thrown, if the query is actually contradictory. □

As an example, consider the application of the transformation on the example CNF. It yields

$$
\begin{gathered}
(60000ms < (Duration([max\{a.val\text{-}from,\ b.val\text{-}from\},\ min\{a.val\text{-}to,\ 2021\text{-}01\text{-}01\}]) \\
\wedge\ (b.val\text{-}from \leq a.val\text{-}to) \\
\wedge\ (e.val\text{-}to \leq a.val\text{-}to) \wedge (e.val\text{-}to \leq b.val\text{-}to) \\
\wedge\ (a.val\text{-}from \leq e.val\text{-}to) \wedge (a.val\text{-}from \leq b.val\text{-}to) \wedge (e.val\text{-}from \leq a.val\text{-}to) \wedge (e.val\text{-}from \leq \\
b.val\text{-}to)
\end{gathered}
$$

$$
\begin{gathered}
\wedge\ (b.val\text{-}from \leq e.val\text{-}to) \\
\cancel{\wedge\ (a.val\text{-}from \leq a.val\text{-}to) \wedge (e.val\text{-}from \leq e.val\text{-}to) \wedge (b.val\text{-}from \leq b.val\text{-}to) \wedge (2020\text{-}06\text{-}01 \leq 2021\text{-}01\text{-}01)} \\
\wedge\ (a.val\text{-}from \leq 2021\text{-}01\text{-}01) \wedge (2020\text{-}06\text{-}01 \leq a.val\text{-}to) \\
\wedge\ (b.val\text{-}from \leq 2021\text{-}01\text{-}01) \wedge (2020\text{-}06\text{-}01 \leq b.val\text{-}to) \\
\wedge\ (e.val\text{-}from \leq 2020\text{-}06\text{-}01) \wedge (2020\text{-}06\text{-}01 \leq e.val\text{-}to)
\end{gathered}
$$

Newly added constraints are colored green, deleted ones are striked through. $(2020\text{-}06\text{-}01 \leq a.val\text{-}to)$ follows from $(2020\text{-}06\text{-}01 \leq e.val\text{-}to)$ and $(e.val\text{-}to \leq a.val\text{-}to)$. Analogously, $(2020\text{-}06\text{-}01 \leq b.val\text{-}to)$ is implied by $(2020\text{-}06\text{-}01 \leq e.val\text{-}to)$ and $(e.val\text{-}to \leq b.val\text{-}to)$.

It is important to note that not every contradictory query is recognized as such by Algorithm 12. This is because disjunctive CNF clauses and singleton clauses with MIN/MAX expressions are not considered, although they may imply contradictions as well.

Because of the newly added constraints, some disjunctive clauses may have become redundant. The transformation Temporal Subsumption deals with this issue.

6.1.8. Temporal Subsumption

The concept of subsumption has already been introduced in Section 6.1.3. In Syntactic Subsumption, a comparison c_1 only subsumes another comparison c_2 if the two clauses are identical. Temporal Subsumption is more sophisticated.

In this transformation, a comparison $c_1 = (s\ comp_1\ l_1)$ subsumes another comparison $c_2 = (s\ comp_2\ l_2)$, if $c_1 \Rightarrow c_2$. Here, s is a time selector, while l_1 and l_2 are literals. Besides that, the transformation works analogously to Syntactic Subsumption, as shown in Algorithm 8.

Lemma 11. *For a CNF ϕ and a formula $\phi' :=$*temporalSubsumption(ϕ)*, ϕ' is a CNF and $\phi' \equiv \phi$.*

Proof. Cf. proof of Lemma 6. □

Syntactic Subsumption is a special case of Temporal Subsumption. Temporal Subsumption is applied after the inference transformation (Section 6.1.7), as this transformation may add further clauses suitable for Temporal Subsumption to the CNF.

Applying Temporal Subsumption to the running example does not change it, as it does not contain disjunctive clauses. Singleton clauses can not subsume each other after the application of InferBounds, as this transformation only leaves one comparison with a literal for each selector.

Now, all transformations employed in Query Rewriting have been introduced.

6.1.9. Summary of Query Transformations

Algorithm 18 puts all transformations together to describe the whole pipeline.

Algorithm 18 QueryTransformation

```
1:  procedure Query transformation pipeline(cnf)
2:      cnf' = Normalize(cnf)
3:      cnf' = UnfoldMinMax(cnf')
4:      cnf' = SyntacticSubsumption(cnf')
5:      cnf' = TrivialTautologies(cnf')
6:      cnf' = TrivialContradictions(cnf')
7:      if TrivialContradictions threw error then
8:          query contradictory, throw error
9:      cnf' = AddTrivialConstraints(cnf')
10:     cnf' = InferBounds(cnf')
11:     if InferBounds threw error then
12:         query contradictory, throw error
13:     cnf' = TemporalSubsumption(cnf')
14:     return cnf'
```

Lemma 12. *For a CNF ϕ, Algorithm 18 returns either a CNF ϕ' or an error. If an error is returned, ϕ is contradictory. Otherwise, $\phi' \equiv \phi$.*

Proof. Follows directly from Lemmas 3, 5, 6, 7, 8, 9, 10 and 11. □

The next Section concludes the discussion of query rewriting with few remarks on the implementation.

6.1.10. Implementation of Query Rewriting

The CNF to transform is represented as a POJO in Gradoop (`CNF.java`). Thus, the implementations of all transformations here are simple java classes, e.g. `SyntacticSubsumption.java`. All of them implement an interface `QueryTransformation.java` that defines a method `CNF transformCNF(CNF cnf)`. Hence, the pipeline is easily extendable by further transformations.

6.2. Selectivitiy Estimations

Another approach to optimize query processing was already discussed in Section 5.1.4. Junghanns et al. [28] implemented a query planning method to find a good join sequence for the query execution. Algorithm 2 sums up this approach. However, for the estimation of cardinalities in line 13 of Algorithm 2, an important aspect is still missing. As pointed out in Section 5.1.4, result cardinality estimation relies on estimations of predicate selectivities. This means that for every predicate θ', an estimation $sel(\theta')$ with $0 \leq sel(\theta') \leq 1$ must be computed. Small values of $sel(\theta')$ indicate a restrictive predicate.

The implementation of the EPGM pattern matching operator by Junghanns et al. [28] simply sets $sel(\theta')$ to 1 for every predicate θ. Thus, predicate selectivities are in effect ignored and cardinality estimations only rely on statistics about label distributions and structural properties. Obviously, a reasonable implementation of predicate selectivity estimation improves the estimations and hence the speed of query processing.

In the following, an implementation of selectivity estimation is described. First, the general approach is introduced in Section 6.2.1. As graphs are typically very large, estimations are computed using samples. The sampling method is discussed in Section 6.2.2. In the remaining, selectivity estimations for different types of comparisons (Sections 6.2.3 - 6.2.6) and structural properties of the graph (Section 6.2.8) are presented. Moreover, predicate selectivities motivate another query transformation which is introduced in Section 6.2.7.

6.2.1. Estimating the Selectivity of a CNF

Query predicates θ are always transformed into a CNF by Gradoop. The predicates to be estimated are CNFs, too, as they are conjunctions of clauses of θ. In order to be able to handle CNFs, a naive assumption is made: every comparison in a CNF is assumed to be independent of all other comparisons. It is obvious that this does not actually hold in general.

For the following discussion, let θ be a CNF consisting of disjunctive clauses $D_1, ..., D_n$, i.e. $\theta = D_1 \wedge D_2 \wedge ... \wedge D_n$. Moreover, every clause D_i is a disjunction of $|D_i|$ comparisons $c_1, ..., c_{|D_i|}$, meaning that $D_i = c_{i,1} \vee c_{i,2} \vee ... \vee c_{|D_i|}$.

Ideally, one could compute a probability function $\mathcal{P}^{comp}$ that assigns every comparison its actual probability of being true for a random suitable graph embedding. The selectivity of a comparison $c_{i,j}$ would then simply be $\mathcal{P}^{comp}(c_{i,j})$. Similarly, probability functions $\mathcal{P}^{clause}$ and $\mathcal{P}^{CNF}$ for clause and CNF selectivity are desirable. Of course, these exact probabilities are not available. Hence, reasonable estimations $\hat{\mathcal{P}}^{comp}$, $\hat{\mathcal{P}}^{clause}$ and $\hat{\mathcal{P}}^{CNF}$ need to be found.

Suppose for now, the estimated probability $\hat{\mathcal{P}}^{comp}$ is already available. Note that the computation of $\hat{\mathcal{P}}^{comp}$ is described in the Sections 6.2.3 - 6.2.6. As a clause consists of comparisons, $\hat{\mathcal{P}}^{comp}$ is used to determine $\hat{\mathcal{P}}^{clause}$.

It is clear that $c_{i,1} \vee c_{i,2} \vee ... \vee c_{i,|D_i|} \Leftrightarrow (c_{i,1}) \vee (c_{i,2} \vee ... \vee c_{i,|D_i|})$ and $\mathcal{P}^{clause}((c_{i,1})) = \mathcal{P}^{comp}(c_{i,1})$. The addition rule for probabilities in combination with the independence assumption yields the recursive estimation

$$\hat{\mathcal{P}}^{clause}(D_i) = \hat{\mathcal{P}}^{clause}(c_{i,1} \vee c_{i,2} \vee ... \vee c_{i,|D_i|}) = \hat{\mathcal{P}}^{comp}(c_{i,1}) + \hat{\mathcal{P}}^{clause}(c_{i,2} \vee ... \vee c_{i,|D_i|}) - (\hat{\mathcal{P}}^{comp}(c_{i,1}) * \hat{\mathcal{P}}^{clause}(c_{i,2} \vee ... \vee c_{i,|D_i|})) \tag{6.16}$$

From the independency assumption for comparisons it follows that clauses are independent from each other, too. Hence, determining $\hat{\mathcal{P}}^{CNF}$ is straightforward.

$$\hat{\mathcal{P}}^{CNF}(\theta) = \hat{\mathcal{P}}^{CNF}(C_1 \wedge C_2 \wedge ... \wedge C_n) = \prod_{i=1}^{n} \hat{\mathcal{P}}^{clause}(C_i) \tag{6.17}$$

Thus, estimating a CNF selectivity is reduced to estimating the selectivity of single comparisons, i.e. determining $\hat{\mathcal{P}}^{comp}$. The following chapters adress this problem.

6.2.2. Graph Sampling

Computing estimations for comparison selectivities requires knowledge about the distributions of property values and intervals in the data graph. Such data graphs can be arbitrarily large, so that the computation of statistics on it requires much time. Hence, the estimations introduced in the Sections 6.2.3 - 6.2.6 are computed using only a sample of the data graph.

Several graph sampling methods are implemented in Gradoop. They are described in detail by Gomez et al. [20]. However, none of these methods is directly applicable here. The aim of sampling here is to acquire a sample set of vertices and edges for every possible label. These sets are then used to compute statistics about property and interval distributions with respect to a given label.

For every vertex and edge label, a reservoir sample (Vitter [67]) of vertices/edges is constructed. In the following, $s_V(l)$ denotes the set of sampled vertices labeled l. Analogously, $s_E(l)$ is defined for edges. The maximum sizes of reservoir samples must be stated in advance. Here, all reservoir samples are assigned the same maximum size, which is simply denoted by *max_size*. Apart from sampling, the numbers of vertices/edges with a given label are counted for every label. These numbers are denoted $|V_l|$ and $|E_l|$.

This method has two main advantages. Every vertex labeled l has the same chance $\frac{1}{|s_V(l)|}$ of being sampled due to the use of Reservoir Sampling. The same holds for edges, obviously. Moreover, the algorithm only needs one pass through the data.

The major downside is that the computed samples are in no way preserving the structure of the underlying data graph. In fact,
$(\bigcup_{l \text{ vertex label}} s_V(l), \bigcup_{m \text{ edge label}} s_E(m))$ may not even be a well-defined graph. For a sampled edge e it is not guaranteed that its source and target vertices were sampled, too. These disadvantages, however, are acceptable here. In order to estimate the probability that a comparison including one or two graph elements holds, the distributions of property and temporal values among the vertices/edges in question provide a reasonable approximation. Estimating structural properties, e.g. the number of distinct source vertices for edges with a given label, is hard, however. This problem is discussed in Section 6.2.8.

Before, the next Sections demonstrate how comparisons between properties or time stamps can be estimated on the basis of reservoir samples.

6.2.3. Time Stamp Comparisons

In TPGM, every graph element is associated with a transaction and a valid interval described by the four timestamps *tx-from*, *tx-to*, *val-from* and *val-to*. GDL allows to compare them among each other or to time literals. To estimate the selectivity of such comparisons, the distributions of these four time stamps must be estimated. The distributions are estimated separately for every vertex and edge label.

A problem in estimating these time stamps is that the type of their distribution can not be known apriori. E.g., in a social network like Figure 2.2 , the distribution of *tx-from* values of *Person* vertices might look similar to a normal distribution: there may be only few users at the launch of the social network. During the next months, the number of users might increase drastically. Afterwards, the user number might only grow slowly. Hence, the distribution of person's *tx-from* values might resemble a normal distribution. In contrast, the *tx-from* values of *Company* vertices might be very similar among each other. Before the social network is launched, all existing companies might have been automatically added to the knowledge graph. New companies are later only added occasionally, causing the *tx-from* value distribution for cities to resemble an exponential distribution. Thus, for every label, the distributions of the four time stamps can not be assumed to resemble a certain statistical distributions apriori.

For this reason, binning is used to approximate the distributions. For every reservoir sample and each of the four time stamps, 100 equally sized bins are created. E.g., there are 100 bins for each property *tx-from*, *tx-to*, *val-from* and *val-to* for vertices labeled *Person*. Every bin, except the first one, corresponds to an interval $[a, b)$. The first bin represents an interval $(-\infty, b)$ with b the biggest value from the sample that falls into the first bin. The last bin stands for an interval $[a, \infty)$, where a is the smallest value from the sample that falls into the last bin. An interval $[a_i, b_i)$ for the i-th bin $(2 \leq i \leq 99)$ is then simply created by setting $a_i = a_{i-1}$ and $b_i = a_{i+1}$ It is clear that any type of distribution can be approximated by bins. Hence, they are well suited for computing statistics on interval time stamps here.

If a value occurs more than once in the sample, several bins can be equal. To give a simple example for this phenomenon, suppose that the sampled values are $\{1, 2, 3, 3, 3\}$. If one were to create 5 equally sized bins of them, three of the bins would be identical. In the Gradoop implementation of TPGM, graph elements without transaction/valid interval have a corresponding *from* value of `Long.MIN\_VALUE` and a corresponding *to* value of `Long.MIN\_VALUE`. Thus, bins can created from such elements, too. Because of this implementation detail, identical bins are not unlikely. Additionally, these values further motivate the use of bins: for the same label, there can be elements that are not valid anymore (i.e. have a reasonable *val-to* value) and such that are still valid, i.e. having `Long.MIN\_VALUE` as their *val-to* value. Since the difference between `Long.MIN\_VALUE` and reasonable *val-to* values is huge, this distribution can hardly be approximated by any other method than binning. The same argumentation holds for *tx-to* values, obviously.

In the following discussion, $\mathfrak{B}_{V_l}^{\textit{tx-from}}$ denotes the set of bins for the *tx-from* value of vertices labeled l. Sets of bins for edges and/or the other three time stamps are denoted analogously. For a given labeled query variable a, its label is given by $\tau(a)$. Unlabeled query variables ($\tau(a) = \epsilon$) are treated differently, as pointed out later. The notation $\mathfrak{B}[i]$ is used to refer to the i-th bin of some set of bins $\mathfrak{B}$.

$|\mathfrak{B}|$ is the number of bins in a set of bins $\mathfrak{B}$.

For the following estimations, values need to be found in sets of bins. Two functions are defined that simplify the necessary notation.

Definition 28 (Leftmost Bin, Rightmost Bin)**.** *Given a set of bins* $\mathfrak{B}$ *and a value* x*, the* leftmost bin in $\mathfrak{B}$ for x*, denoted* leftmost($\mathfrak{B}$,x) *is defined as*

$$\textit{leftmost}(\mathfrak{B}, x) = \mathfrak{B}[i],\ i = \min_{1 \leq j \leq |\mathfrak{B}|} x \in \mathfrak{B}[j]$$

Analogously, the rightmost bin in $\mathfrak{B}$ for x *is given by*

$$\textit{rightmost}(\mathfrak{B}, x) = \mathfrak{B}[i],\ i = \max_{1 \leq j \leq |\mathfrak{B}|} x \in \mathfrak{B}[j]$$

The easiest case of time stamp comparisons is the comparison between a selector like *a.tx-from* and a literal l, e.g. l =*2020-01-01*, where a's label is given ($\tau(a) \neq \epsilon$).
Let *selector* $\in$ {*tx-from, tx-to,val-from, val-to*}. Then, the following estimations are carried out.

$$\hat{\mathcal{P}}(a.selector < l \mid \tau(a) \neq \epsilon) = \frac{leftmost(\mathfrak{B}^{selector}_{\tau(a)}, l)}{|\mathfrak{B}^{selector}_{\tau(a)}|} \tag{6.18}$$

$$\hat{\mathcal{P}}(a.selector \leq l \mid \tau(a) \neq \epsilon) = \frac{rightmost(\mathfrak{B}^{selector}_{\tau(a)}, l)}{|\mathfrak{B}^{selector}_{\tau(a)}|} \tag{6.19}$$

Both estimations only yield reasonable probabilities if not all sampled values of *a.selector* are of the same value. If all sampled values are of the same value, $leftmost(\mathfrak{B}^{selector}(a), l) = 1/|\mathfrak{B}^{selector}(a)|$ while $rightmost(\mathfrak{B}^{selector}(a), l) = 1$.

Estimations where a is not labelled are more complicated, since they must consider all possible sets of bins suitable for a. Additionally, not all sets of bins are equally important as not all labels are equally frequent. Thus, a simple weight function w is defined for a set of vertex bins:

$$w(\mathfrak{B}^{selector}_{l}) = \frac{|V_l|}{|V|}$$

The weight function for edge bins is defined analogously. Now, the estimations given above can be modified to compute estimates for non-labelled vertices a.

$$\hat{\mathcal{P}}(a.selector < t \mid \tau(a) = \epsilon) = \sum_{l\ vertex\ label} w(\mathfrak{B}^{selector}_{l}) * \frac{leftmost(\mathfrak{B}^{selector}_{l}, t)}{|\mathfrak{B}^{selector}_{l}|} \tag{6.20}$$

$$\hat{\mathcal{P}}(a.selector \leq t \mid \tau(a) = \epsilon) = \sum_{l\ vertex\ label} w(\mathfrak{B}^{selector}_{l}) * \frac{rightmost(\mathfrak{B}^{selector}_{l}, t)}{|\mathfrak{B}^{selector}_{l}|} \tag{6.21}$$

Because every value can be sorted in at least one bin, $1 \geq \hat{\mathcal{P}}(a.selector \leq l) > \hat{\mathcal{P}}(a.selector < l) > 0$ for all possible inputs *a.selector*, l. All estimations for $>$ and $\geq$ can now be simply defined using Equations 6.18, 6.19, 6.20 and 6.21.

$$\hat{\mathcal{P}}(a.selector > l) = 1 - \hat{\mathcal{P}}(a.selector \leq l) \tag{6.22}$$

$$\hat{\mathcal{P}}(a.selector \geq l) = 1 - \hat{\mathcal{P}}(a.selector < l) \tag{6.23}$$

Regardless of a being labelled or not, for most literals l it is very unlikely that $a.selector = l$. Hence, the corresponding estimations are set to constants.

$$\hat{\mathcal{P}}(a.selector = l) = 10^{-3} \tag{6.24}$$

$$\hat{\mathcal{P}}(a.selector \neq l) = 1 - \hat{\mathcal{P}}(a.selector = l) \tag{6.25}$$

Selectors for interval boundaries can also be compared among each other. Let a and b be labelled query variables ($a \neq b$) and $sel_a, sel_b \in \{$*tx-from, tx-to,val-from, val-to*$\}$. Estimating comparisons like $a.sel_a$ *comp* $b.sel_b$ using bins is significantly more difficult. Hence, for this kind of estimation, it is assumed that the four temporal properties are normally distributed. This means that for every reservoir and temporal property, the mean and variance must be computed.

The reason for this assumption is that the difference between two normally distributed random variables $X_1 \sim \mathcal{N}(m_1, \sigma_1^2)$ and $X_2 \sim \mathcal{N}_2(m_2, \sigma_2^2)$ is normally distributed, too: $X_1 - X_2 \sim (m_1 - m_2, \sigma_1^2 + \sigma_2^2)$. Thus, $\hat{\mathcal{P}}(a.sel_a \leq b.sel_b)$ can be simply estimated as

$$\hat{\mathcal{P}}(a.sel_a \leq b.sel_b) = \hat{\mathcal{P}}((a.sel_a - b.sel_b) \leq 0) \tag{6.26}$$

where $\hat{\mathcal{P}}((a.sel_a - b.sel_b) \leq 0)$ is computed with the cumulative probability function for a normal distribution. Moreover, estimations on how many elements have `Long.MIN_VALUE` or `Long.MAX_VALUE` are taken into account. Estimations for equality predicates are obtained analogously, using the density function of the normal distribution:

$$\hat{\mathcal{P}}(a.sel_a = b.sel_b) = \hat{\mathcal{P}}((a.sel_a - b.sel_b) = 0) \tag{6.27}$$

Estimations for the remaining comparators ($<$, $\neq$, $\geq$, $>$) can then be computed using Equations 6.26 and 6.27 in a straightforward way, e.g.

$$\hat{\mathcal{P}}(a.sel_a < b.sel_b) = \hat{\mathcal{P}}(a.sel_a \leq b.sel_b) - \hat{\mathcal{P}}(a.sel_a = b.sel_b) \tag{6.28}$$

The normal distribution is a continuous probability distribution, while time stamps are discrete values. However, this is not problematic here, as Equation 6.26 only uses a cumulative probability. Exact probability densities are needed for estimating the probability of equality between two stamps. However, because equality between two time stamps is, in general, considered very unlikely, it is acceptable that Equation 6.27 might yield a very small value.

In order to estimate comparisons between one or two unlabeled query variables, weights for different bin sets, analogously to Equations 6.20 and 6.21, are employed. The formulas are omitted here for the sake of brevity.

As shown in Section 6.1.2, many types of MIN/MAX comparisons are reformulated to simple time stamp comparisons during the query transformation pipeline. For the remaining MIN/MAX comparisons, estimating their selectivity would be possible with some overhead. Every MIN/MAX comparison can be unfolded to a boolean formula including only simple time stamp comparisons, as shown in detail by Lemma 4. Such a formula could be transformed to a CNF and then estimated with the methods introduced above. Given this extra complexity and the fact that many MIN/-MAX comparisons are already eliminated at this point, the remaining MIN/MAX comparisons' probability estimations are ignored. This means, they are always set to 1.

Besides comparing time stamps, GDL also allows the comparison of interval durations. The estimations for such terms also makes use of normal distributions.

6.2.4. Duration Comparisons

Statistics about durations are computed based on the reservoir samples, too. It is assumed that durations are normally distributed. Hence, for every reservoir sample like $s_V(l)$ four values $\mu_V^{dur(tx)}(l)$, $\sigma_V^{dur(tx)}(l)$, $\mu_V^{dur(val)}(l)$ and $\sigma_V^{dur(val)}(l)$ are computed as follows.

$$\mu_V^{dur(tx)}(l) = \frac{1}{|s_V(l)|} \sum_{a \in s_V(l)} \textit{tx-duration(a)}$$

with

$$\textit{tx-duration(a)} = \begin{cases} \textit{a.tx-to} - \textit{a.tx-from} & \textit{if a.tx-to} < \texttt{Long.MAX\_VALUE} \\ \textit{stamp(now)} - \textit{a.tx-from} & \textit{else} \end{cases}$$

Here, *tx-duration(a)* ensures that the duration is never infinite. Then, the standard deviation $\sigma_V^{dur(tx)}(l)^2$ can be computed using $\mu_V^{dur(tx)}(l)$.

The values $\mu_V^{dur(val)}(l)$ and $\sigma_V^{dur(val)}(l)^2$ are, of course, computed analogously. For every reservoir sample like $s_V(l)$ transaction and valid durations are assumed to be normally distributed random variables $D_V^{tx}(l) \sim (\mu_V^{dur(tx)}(l), (\sigma_V^{dur(tx)}(l))^2)$ and $D_V^{val}(l) \sim (\mu_V^{dur(val)}(l), (\sigma_V^{dur(val)}(l))^2)$. These distributions are used to estimate probabilities in a straightforward way, e.g.

$$\hat{\mathcal{P}}(\textit{duration(a.tx)} \leq d \mid \tau(a) \neq \epsilon) = \mathcal{P}(D_V^{tx}(\tau(a)) \leq d) \tag{6.29}$$

For the same reasons as in Section 6.2.3, the continuous normal distribution is acceptable for discrete durations.

As already mentioned, the difference between two normally distributed random variables is normally distributed again. Hence, estimations for comparisons between two variable durations can be easily computed, e.g.

$$\hat{\mathcal{P}}(\textit{duration(a.tx)} \leq \textit{duration(b.tx)} \mid \tau(a) \neq \epsilon \neq \tau(b)) = \mathcal{P}((D_V^{tx}(\tau(b)) - D_V^{tx}(\tau(b))) \leq 0) \tag{6.30}$$

Comparisons involving non-labeled elements are estimated using weights in the same way as in Section 6.2.3. Equation 6.31 provides an example for a comparison between two non-labeled vertices a and b.

$$\hat{\mathcal{P}}(duration(a.tx) \leq duration(b.tx) \mid \tau(a) = \tau(b) = \epsilon) = \sum_{l\ vertex\ label} \sum_{m\ vertex\ label} (\frac{|V_l|}{|V|}\frac{|V_m|}{|V|}\mathcal{P}((D_V^{tx}(l) - D_V^{tx}(m)) \leq 0)) \tag{6.31}$$

The selectivities of all types of temporal comparisons can now be estimated. In the following two chapters estimations for comparisons between property values are discussed.

6.2.5. Numerical Property Comparisons

Numeric Property Values (e.g. *Age* in the graph in Figure 2.2) are treated very similar to durations. Just like them, they are assumed to be normally distributed. The only difference is that not every graph element is guaranteed to have a non-NULL value for a certain property. Thus, for every reservoir sample (e.g. $s_V(l)$) and property p, $ratio_p(s_V(l)) := \frac{|\{x \in s_V(l) | \kappa(x,p) \neq NULL\}|}{|s_V(l)|}$ must be computed.

Similar as for durations, random variables $P_V^p(l)$ for every reservoir sample $s_V(l)$ (analogously for edges) and property p can be created. Their means and variances are computed using only the elements in $s_V(l)$ that actually havea non-NULL value for property p.

A comparison between some vertex a's property value for p and a literal x can then be estimated as shown exemplary in Equation 6.32. It is always assured that probabilities are strictly greater than zero. The reason for this is the small chance of $ratio_p(s_V(l))$ being 0, even though there are actually l-vertices with property p in the data graph.

$$\mathcal{P}(a.p \leq x) = \max\{ratio_p(s_V(l)) * \mathcal{P}(P_V^p(l) \leq x),\ 10^{-5}\} \tag{6.32}$$

Again, these estimations can be extended to handle comparisons between two property selectors in a natural way. For two labelled vertices a and b and properties p,q, Equation 6.34 estimates the probability that $a.p \leq b.p$ holds.

$$\mathcal{P}(a.p \leq b.q \mid \tau(a) \neq \epsilon \neq \tau(b)) = max\{ratio_p(s_V(l_1)) * ratio_q(s_V(l_2)) * (\mathcal{P}((P_V^p(l_1) - P_V^q(l_2)) \leq 0), 10^{-5}\} \tag{6.33}$$

Furthermore, comparisons involving unlabelled graph elements are estimated analogously to Sections 6.2.3 and 6.2.4. To give an example, Equation 6.34 shows the estimation for $a.p_1 \leq b.p_2$ if a and b are not labelled.

$$\mathcal{P}(a.p \leq b.q \mid \tau(a) = \tau(b) = \epsilon) = max\{\sum_{l\ vertex\ label} \sum_{m vertex\ label} \frac{|V_l|}{|V|} * \frac{|V_m|}{|V|} * ratio_p(s_V(l)) * ratio_q(s_V(m)) * (\mathcal{P}((P_V^p(l) - P_V^q(m)) \leq 0),\ 10^{-5}\} \tag{6.34}$$

This completes the discussion of numeric property values. However, property values can be non-numeric, i.e. categorical, too.

6.2.6. Categorical Property Comparisons

Categorical Properties can only be compared using $=$ or $\neq$ as there is no order on them. All other comparisons are estimated to hold with probability 0.

As before, the distribution of these property values are estimated based on the reservoir samples. For this purpose, let $s_V^{p=x}(l) \subseteq s_V(l)$ denote those elements of $s_V(l)$ where the value of property p is equal to x. Then, a comparison $a.p = x$, where a is a labeled graph element and x a literal, is easy to handle.

$$\hat{\mathcal{P}}(a.p = x \mid \tau(a) \neq \epsilon) = max\{ratio_p(s_V(\tau(a))) * \frac{|s_V^{p=x}(\tau(a))|}{|s_V(\tau(a))|}, 10^{-5}\} \tag{6.35}$$

Other than in the Sections before, it is not reasonable to simply assume that every equality is rather unlikely. The range of a categorical property value might be very small, e.g. for fields like gender or nationality. Estimations for inequalities are defined using Equation 6.35

$$\hat{\mathcal{P}}(a.p \neq x) = max\{1 - \hat{\mathcal{P}}(a.p = x), 10^{-5}\} \tag{6.36}$$

Again, these comparisons can be straightforwardly adjusted to non-labeled vertices, e.g.

$$\hat{\mathcal{P}}(a.p = x \mid \tau(a) = \epsilon) = \max(\sum_{l\ vertex\ label} \frac{|V_l|}{|V|} * ratio_p(s_V(l)) * \frac{|s_V^{p=x}(l)|}{|s_V(l)|}, 10^{-4}) \tag{6.37}$$

The estimation for equality between two property selectors is given in Equation 6.38. Here, $range_l(p)$ is the set of values x where $|s_V^{p=x}(l)| > 0$.

$$\begin{aligned} \hat{\mathcal{P}}(a.p = b.q \mid \tau(a) \neq \epsilon \neq \tau(b)) = \max(\sum_{x \in range(p) \cup range(q)} \\ ratio_p(s_V(\tau(a))) * ratio_q(s_V(\tau(b))) * \frac{|s_V^{p=x}(\tau(a))|}{|s_V(\tau(a))|} * \frac{|s_V^{q=x}(\tau(b))|}{|s_V(\tau(b))|}, 10^{-4}) \end{aligned} \tag{6.38}$$

If there are one or two unlabeled elements in such a comparison, the estimation can be extended analogously to Equation 6.37.

All types of comparisons can now be assigned a probability estimation. Hence, for every possible CNF, $\hat{\mathcal{P}}^{CNF}$ can be computed. The comparison selectivity estimations are used to apply another transformation on the query.

6.2.7. Clause Reordering

In its Gradoop implementation, Algorithm 2 always checks CNF clauses from left to right. Thus, the number of expected clauses to check can be minimized by sorting the clauses according to their estimated selectivity. The most selective clauses should be encountered first in order to filter out embeddings as soon as possible.

Disjunctive clauses are processed from left to right, too. Within them, less selective comparisons should be checked first, as one positively evaluated comparison is enough to satisfy the clause. Hence, the expected number of comparisons to check the clause is minimized.

Algorithm 19 sums up this CNF transformation.

Algorithm 19 Clause Reordering

1: **procedure** ReorderClauses(cnf)
2: $cnf' =$ empty CNF
3: **for** $clause = (c_1, ..., c_{|clause|}) \in cnf$ **do**
4: $clause' =$ reorder $clause'$ acc. to estimateSel(c_i) (descending)
5: $cnf' = cnf' \wedge clause'$
6: $cnf'' =$ reorder cnf' acc. to estimateSel(C_i)(cnf'=$C_1 \wedge ... \wedge C_{|cnf'|}$) (ascending)
7: **return** cnf'

Lemma 13. *For a CNF ϕ and a formula $\phi' :=$reorderClauses(ϕ), ϕ' is a CNF and $\phi' \equiv \phi$.*

Proof. As only clauses and comparisons within clauses are reordered by Algorithm 19, it is clear that ϕ' is a CNF again and $\phi \equiv \phi'$. □

This transformation was not introduced in Section 6.1, as it is not part of the pipeline described there. Instead, the output of the query rewriting pipeline is the basis for all selectivity estimations. Algorithm 19 can only be applied after these estimations are available.

6.2.8. Structural Properties

The join cardinality estimation approach introduced in Section 5.1.4 is based on selectivity of query predicates as well as on structural properties of the data graph. Estimations for query predicate selectivities have been pointed out above, those for structural properties are yet missing.

These structural properties are, as mentioned in Section 5.1.4:

- the overall number of vertices ($|V|$) and edges ($|E|$)
- the number of vertices or edges labelled l ($|V_l|$ and $|E_l|$ respectively) for every label l
- the cardinalities $|Values(\mathcal{R}, a.ID)|$ for an edge join of some vertex variable a

Lukas Christ
3756213

$|V|$ and $|E|$ as well as $|V_l|$ and $|E_l|$ for every label l are computed exactly during sampling. To achieve this, the respective graph elements are simply counted, even though only a fraction of them may be sampled.

$|\mathit{Values}(\mathcal{R}, a.ID)|$ is more problematic. $\mathcal{R}$ is the relation containing the edge e to join, a is either e's source or target vertex. W.l.g., assume that a is e's source vertex. Then, $|\mathit{Values}(\mathcal{R}, a.ID)|$ is the number of distinct source vertices of type a for an edge of type e. Here, the *type* of a vertex (analogously, an edge) is the set of vertices with the same label, if the vertex is labeled. Otherwise, the type of a is just the set of all vertices.

Hence, for every combination of a vertex label l_v and an edge label l_e, the number of distinct source and target vertices labelled l_v for edges labelled l_e is needed. Furthermore, for every edge label l_e, the number of distinct source and target vertices overall, regardless of the vertex label, must be computed. All these values can be computed exactly on the graph using the graph statistics provided by Gradoop. In order to save time, however, in the implementation described here, they are only estimated based on the Reservoir Samples. E.g., for an edge e and its source vertex a, labeled l_a, $|\mathit{Values}(\mathcal{R}, a.ID)|$ (where e is specified in $\mathcal{R}$) is computed as

$$|\mathit{Values}(\mathcal{R}, a.ID)| = |\{(u, v) \in s_E(\tau(e)) \mid u \in V_{l_a}| * \frac{|E_{\tau(e)}|}{|s_E(\tau(e))|} \tag{6.39}$$

Since edge sampling does not preserve structural properties like vertex degrees [38], this estimation is not accurate. Its advantage is that it can be computed fast and provides reasonable values in many cases. As the focus of the estimations implemented here was on estimations for property values, no other method for structural properties is provided. However, the implementation allows for better approaches being added in future work, as the next Section shows.

6.2.9. Implementation of Estimations

The estimations introduced above can be seen as a reference implementation. They could be replaced by other implementations of the abstract class `TemporalGraphStatistics.java` in the future. This class prescribes several methods to estimate probabilities and structural properties. The class
`BinningTemporalGraphStatistics.java` extends it and implements the methods described in this Section. Because of `TemporalGraphStatistics.java` being employed in query planning, the query planning component is different from the EPGM implementation.

`TemporalGraphStatistics.java` forces its sub-classes to provide four types of factory methods for creating instances:

- `fromGraph(TemporalGraph g)`: create statistics for the graph. Consider every property.
- `fromGraph(TemporalGraph g, Set<String> numericalProperties,`

  `Set<String> categoricalProperties)`: create statistics for the graph, but consider only the specified properties.

- `fromGraphWithSampling(TemporalGraph g, int sampleSize)`: create statistics for the graph that are guaranteed to be based on a sample. The sample should have size `sampleSize`.
- `fromGraphWithSampling(TemporalGraph g, int sampleSize, Set<String> numericalProperties, Set<String> categoricalProperties)`: create statistics for the graph based on a sample of size `sampleSize`, considering only the specified properties.

In the API (see Appendix B), the user is free to choose any statistics implementation. It is also possible to call the operator without using any statistics. In this case, a dummy implementation of `TemporalGraphStatistics.java` is employed that sets every estimation to 1.

7. Evaluation

In the following, the implemented Pattern Matching operator is evaluated. Section 7.1 describes the technical details of the Evaluation. In Section 7.2, the data sets used are introduced. Finally, the results are presented and discussed in Section 7.3.

7.1. Evaluation Setup

Evaluation is performed on a share-nothing cluster. It consists of 16 physical workers, each of them handling up to 6 parallel threads. Thus, the degree of parallelity is 6 for one worker, 12 for two,..., and 96 if all 16 workers are used.

1 GBit/s Ethernet connections link the workers among each other. Every worker is equipped with an Intel Xeon E5-2430 v2 CPU (2,5 GHz) and 48 GB of RAM. For the execution, the Flink version 1.9.0 and Hadoop in the version 2.6.0 was used.

The program is run three times for every query. The mean duration of these runs is then considered the query's run time. A run of a query comprises reading the data graph from the source, executing the matching and writing the results back to a sink.

7.2. Evaluation Data Sets

Two data sets are used for the evaluation: the real-world citibike data set (Section 7.2.1) and the synthetic LDBC data set (Section 7.2.2).

7.2.1. The Citibike Data Set

Citibike[8] is a bike-sharing company based in New York. They provide data about their customers' bike trips from 2013 on[9].

This data is very homogeneous: every row in the data set describes a trip from one place ('station') in NYC to another. For each trip, a start and end time stamp as well as the trip's duration is provided. Furthermore, some information about the bike (bike id) and the customer is given, e.g. gender and year of birth. Stations have various attributes, including their capacity, id and name. Further attributes for each station are given in a separate JSON[10], e.g. a short name and whether the station has a kiosk.

In Gradoop, an importer for citibike data is already implemented. It offers two options for importing the data. With the option `TRIPS_AS_EDGES`, stations are imported as vertices and trips as edges connecting stations. The option `TRIPS_AS_VERTICES` allows to map trips to vertices. As

[8] `citibikenyc.com`, accessed August 2020
[9] `citibikenyc.com/system-data`, accessed August 2020
[10] `https://gbfs.citibikenyc.com/gbfs/en/station_information.json`, accessed August 2020

the `TRIPS_AS_EDGES` option creates a more intuitive graph, it is chosen for the evaluation. The importer also loads the additional station attributes into station vertices.

There are no explicit transaction and valid intervals in citibike data. However, the trip interval ranging from start of a trip to its end is a natural choice for a trip edge's valid interval. Transaction intervals are not set, i.e. they keep their default range from `Long.MIN_VALUE` to `Long.MAX_VALUE`.

All citibike data sets from 06/2013 to 05/20202 are used to create the evaluation data set. It is stored in Gradoop's `IndexedCSV` format, occupying about 23 GB of memory. To evaluate the operator's behaviour regarding different data set sizes, two smaller data sets are created from the citibike data set. The whole citibike data set is considered having scalability factor (SF)100. Sampling edges with a probability of 10% creates the citibike data set with SF10. By sampling edges with probability 1%, the citibike data set with SF 1 is obtained. Consequently, SF10 takes about 2.3 GB space while SF1's size is about 230 MB.

A problem with the citibike data set is that it is relatively small. As Gradoop is intended to handle big data, the operator must be evaluated on a larger data set, too.

7.2.2. The LDBC Data Set

LDBC-SNB [13] is a synthetic benchmark for big data operations inspired by social networks. Its schema is richer than the citibike schema. Furthermore, it is possible to create large data sets using the provided generator[11].

LDBC SNB graphs are property graphs according to the PGM (cf. Definition 5). There are various types of vertices and edges in LDBC SNB graphs. For the evaluation here, the dynamic parts of the graph are most interesting. Dynamic vertex types are *Person*, *Forum*, *Post*, *Comment* and *Message*, where *Message* is the super-type of *Post* and *Comment*. These vertices are connected via different types of dynamic edges, among them *knows* (between *Person*s), *likes* (from *Person*s to *Message*s) and *replyOf* (from *Comment*s to *Message*s).

All the mentioned dynamic vertex and edge types are equipped with a *creationDate* property of type *DateTime*. This property is used as *val-from* timestamp in the evaluation. The other three time stamps defined by TPGM are not set, i.e. set to their default values. Intervals of graph elements without *creationDate* property are set to default values, too. An exception is the *hasMember* edge from *Forum*s to *Person*s. For this type of edge, the *joinDate* property is used as *val-from* value. The LDBC data generator provides the opportunity to create synthetic data graphs of different sizes. LDBC graphs of scalability factors (SF) 100, 10 and 1 are used for the evaluation. Again, the data is stored in Gradoop's `IndexedCSV` format. SF100 requires ca. 422 GB, SF10 about 42 GB and SF1 4.2 GB of disk space.

[11] `https://github.com/ldbc/ldbc_snb_datagen`, accessed August 2020

7.3. Results

Based on the two data sets, the operator's performance regarding different query complexities (7.3.1) and queries of different selectivity (7.3.2) is evaluated. Furthermore, the runtime for various data sizes and degrees of parallelity is measured (7.3.3). After that, the problems of comparing the TPGM operator with the EPGM operator are discussed in Section 7.3.4.

7.3.1. Query Complexity

Queries consist of a (potentially labelled) graph pattern and a set of constraints. Thus, queries can be of different complexity regarding their pattern as well as their constraints. The complexity of a query is expected to influence the operator's runtime.

First, the complexity of patterns is examined more closely. Pattern complexity determines the number of joins needed to construct the result set (cf. Algorithm 2). Hence, it is reasonable to expect different patterns to affect the run time differently. In order to gain insight into the influence of pattern complexity on the performance, 8 different citibike graph patterns are considered. For every pattern, a query is built by restricting the pattern's edges' *val-from* values, leading to queries c0,...,c7. The restrictions are different for each query. For the more complex queries c3-c7, the temporal restrictions are tuned so that the results of all these queries are of similar size. The queries and their result sizes on the whole citibike data set are listed in Table 7.1. Temporal restrictions are given in round brackets in the *Query* column.

Query	Pattern	Cardinality
$c0$ (-)	`(s1)`	1174
$c1$ (1d)	`(s1)-[t]->(s2)`	65462
$c2$ (1d)	`(s1)-[t]->(s1)`	1215
$c3$ (1d)	`(s1)-[t1]->(s2)<-[t2]-(s3)`	12294676
$c4$ (1d)	`(s1)-[t1]->(s2)-[t2]->(s3)`	12311535
$c5$ (1d)	`(s3)<-[t2]-(s1)-[t1]->(s2)`	12267234
$c6$ (1h)	`(s1)-[t1]->(s2)-[t2]->(s3)<-[t3]-(s1)`	14254172
$c7$ (90m)	`(s1)-[t1]->(s2)-[t2]->(s3)<-[t3]-(s4)`	14633859

Table 7.1.: Citibike query patterns and their result cardinalities on citibike SF100

All of them are executed on the whole citibike data set, using 16 workers, i.e. 96 parallel threads. Figure 7.1 shows the average run time for each query.

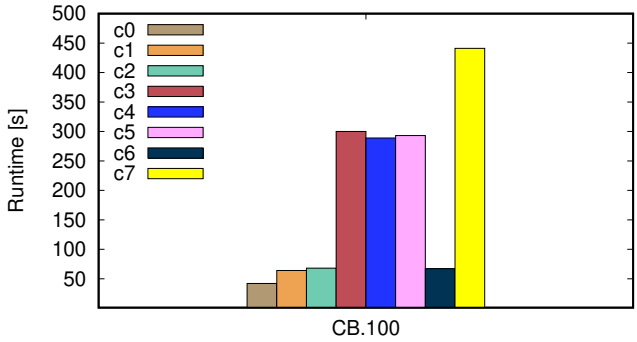

Figure 7.1.: Runtimes of patterns c0...c7 on citibike SF100 (parallelity 96)

Not surprisingly, queries including joins (i.e., all but c0) require more time than c0, the only query without any join. Comparing the queries ranging over one day (c1, c2, c3, c4, c5) indicates that there is indeed a correlation between number of joins and run time. The very similar run times of queries c3, c4 and c5 demonstrate that the complexity, i.e. the number of necessary joins, but not the exact structure of a pattern is relevant for the operator's performance. Furthermore, for the simple queries c1 and c2, the result set size is less important than the pattern complexity. Both queries have nearly identical run times. Here, the intermediate results of c2 are even smaller than those of c1, since only self-loop edges are kept as candidates for t in the first step of c2's execution. As both queries' results are comparatively small, the result sets' sizes may not heavily influence the run time here.

Another indication of the pattern complexity's importance is the run time of c7. As c7 is restricted to just 90 minutes, the number of edge candidates for each edge is way smaller than in the queries c1-c5. Nevertheless, c7 as a more complex query involving three edges takes significantly more time than any of the queries c1-c5. More surprising, c6's run time is rather low, similar to that of c1 and c2. The main reason for this is that the pattern is restricted to one hour, which is the strictest restriction of all queries. Thus, the initial candidate sets for vertices and edges are smaller than for any other query. As $c7$ is restricted to 1.5 hours, the number of candidate edges for $c7$ is about $1.5^3 = 3.375$ times larger than the number of $c6$'s candidate edges. Moreover, because of $(s1)$ appearing two times in $c6$, there is one less vertex candidate set than in $c7$.

The performance for different query patterns is evaluated for LDBC data, too. Patterns similar to those used in the citibike evaluation above are created, resulting in queries l0,...,l7. All *likes* edges are restricted to one day by means of their valid interval. Vertices referring to messages (i.e. posts and comments) are restricted to two days in the same way. Table 7.2 lists all the patterns and their result cardinalites for LDBC SF100.

The queries are executed on LDBC SF100 with parallelity 96. Figure 7.2 visualizes the results.

Query	Pattern	Cardinality
$l0$	`(p1:person)`	448626
$l1$	`(p1:person)-[l:likes]->(m)`	211945
$l2$	`(p1:person)-[l1:likes]->(m)` `(p2:person)-[l2:likes]->(m)`	12582910
$l3$	`(p:person)-[l:likes]->(c:comment)` `(c)-[r:replyOf]->(post:post)`	76301

$l4$	`(p1:person)-[k:knows]->(p2:person)`
	`(p2:person)-[l1:likes]->(m)`
	`(p1)-[l2:likes]->(m)`
		222350

$l5$	`(p1:person)-[l1:likes]->(c:comment)`
	`(c)-[r:replyOf]->(m)`
	`(p1)-[l2:likes]->(m)`
		608

$l6$	`(p1:person)-[k1:knows]->(p2:person)`
	`(p2)-[l1:likes]->(m)`
	`(p3:person)-[l2:likes]->(m)`
	`(p1)-[k2:knows]->(p3)`
		22141686

$l7$	`(p1:person)-[l1:likes]->(c1)`
	`(c1)-[r1:replyOf]->(m)`
	`(c2)-[r2:replyOf]->(m)`
	`(p1)-[l2:likes]->(c2)`
		622

Table 7.2.: LDBC query patterns and their result cardinalities

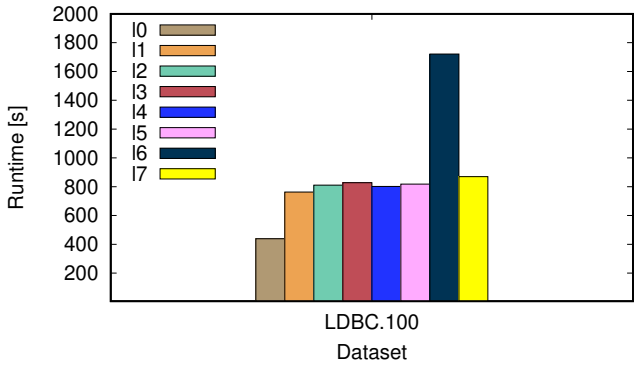

Figure 7.2.: Runtimes of patterns l0...l7 on LDBC SF100 (parallelity 96)

Analysing them is more complicated than for the homogeneous citibike data set. Not surprisingly, the run times are, in general, longer than for the queries on the much smaller citibike data. As one would expect, l0 requires the least time. Comparing the run times of l6 and l7 to the other queries' run times shows that pattern complexity, i.e. the number of joins, is not the only factor influencing the run time. Once again, the results demonstrate that the size of intermediate results has a crucial impact on the run time. For l6, they are bigger than for any other query, as indicated by l6's result cardinality. The run times of l1 - l5 and l7 are very similar, even though the patterns are of different complexity. These queries have either comparatively small (intermediate) result sets

(l5, l7) or just two (l2, l3) joins. A special case is l4 with more results than l3, l5 and l7 and a more complex pattern than in l2 and l3.

Figure 7.3 combines the results for comparable citibike and LDBC queries. The diagram makes clear that the run times behave similar for both data sets. Furthermore, it is interesting that, even though LDBC (SF100) is about 20 times larger than CB (SF100), the run times do not differ as much. This is mostly due to the fact that for the citibike data set, the result sizes are, in general, not smaller. Thus, only the candidate selection step takes significantly longer for LDBC.

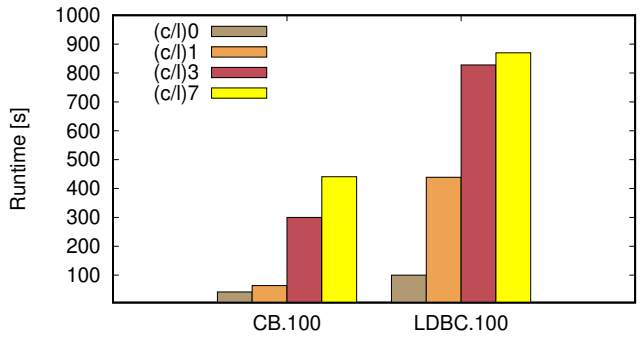

Figure 7.3.: Runtimes of selected patterns on citibike SF100 and LDBC100 (parallelity 96)

Overall, none of the results contradict the intuition that run time mainly depends on the amount of data that needs to be passed through the query plan execution. This, in turn, depends on the number of joins as well as on the number and size of intermediate result embeddings. Hence, it is not possible to predict a query's run time by only considering pattern complexity.

The other aspect of query complexity is the complexity of constraints, which is more elusive than pattern complexity. How a constraint influences the run time depends on several aspects of the constraint: whether it refers to one or two elements, its selectivity and its redundancy.

An experiment shows the impact of a constraint's position in the query plan. A constraint referring only to a single variable can always be evaluated in the first step of Algorithm 2 , i.e. the initial candidate selection. Constraints comparing two different variables can only be processed in the context of a join. Hence, a single variable constraint will reduce the intermediate result size earlier than a join constraint. This will of course affect the run time that mainly depends on the size intermediate results. To demonstrate this effect, three variants c5a, c5b and c5c of query c5 for citibike are created. In c5a, only constraints referring to a single variable are contained, while c5b only includes join constraints (except the temporal constraints restricting the edges' *val-from*). The third variant, c5c, features both types of constraints. All three queries' constraints are tuned to return a comparable number of results. Table 7.3 gives an overview over the queries. The exact queries can be found in Appendix B.1

The three queries are run for citibike SF100. In Figure 7.4, the results of the experiment are visualized.

Query	Constraints	Cardinality
c5a	only single-element constraints	5117478
c5b	only join constraints (besides temporal restrictions for the pattern)	4868340
c5c	both single-element and join constraints	4832533

Table 7.3.: Caption

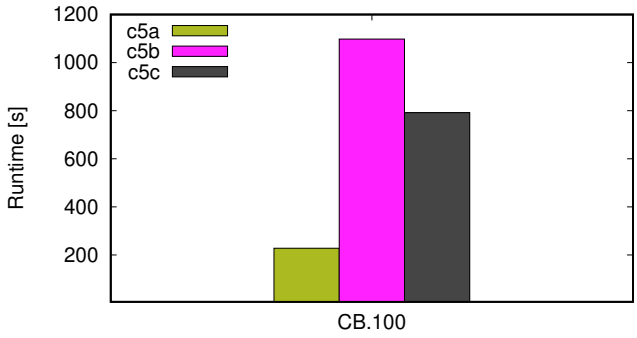

Figure 7.4.: Cardinalities and characteristics of c5a, c5b and c5c on citibike SF100 (parallelity 96)

Not surprisingly, c5a has the lowest run time. In c5a, the intermediate results are the smallest as all the constraints are already checked in the first step of Algorithm 2. Because c5b has only join constraints (except the constraints restricting the edge's valid from values), it has the longest run time. The reason for this is that c5b's intermediate results are the largest among the three queries due to less restrictive single element constraints. Consistent with this reasoning, c5c's performance is better than c5b's but worse than c5a's. It is clearly demonstrated by this experiment, that the type of constraints is an important factor of a query's run time. Hence, the query transformation inferring additional single element constraints (cf. Section 6.1.7) is justified.

The number of constraints influences a query's performance, too, as every constraint must be checked on possible results during query processing. However, the impact of the number of constraints depends on the type of the constraints (as shown in the experiment above), their selectivity and on their redundancy. To demonstrate the latter aspect, another experiment is conducted. For every variant of c5 introduced above, a redundant version is created by doubling the number of constraints. The newly introduced constraints do not change the results. Furthermore, every newly added constraint corresponds to a constraint already present in the query and is checked in the same step as the corresponding constraint. For example, given a comparison `t1.val.lengthAtMost(Hours(10))`, a redundant constraint `t1.val.lengthAtLeast(Seconds(0))` is added to the query. The whole queries can be found in the Appendix. It is ensured that the redundant constraints are not removed by a query transformation. The queries are executed using 96 parallel threads. Figure 7.5 shows the experiment's results by comparing the run time of the original queries c5a, c5b and c5c with their redundant versions on citibike SF100.

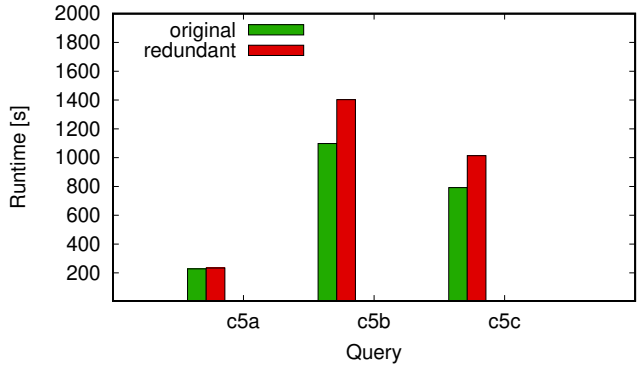

Figure 7.5.: Runtimes for c5a, c5b and c5c, original and redundant version (parallelity 96)

Obviously, the additional constraints increase the run time. This is not surprising as for every result embedding, every constraint has to be checked. However, the impact of redundant constraints is not always drastic here. For c5a, the run time only increases from 228 to 234 seconds (2,6%). All additional constraints in q5a can be handled by the estimation methods introduced in Section 6.2. Hence, the redundant constraints are always checked last and thus only for actual result candidates due to the query reordering transformation (cf. Section 6.2.7). The other two queries are slowed down more notably. There may be two reasons for this: first, the estimation component can not handle the the complex duration constraints introduced here. Hence, a redundant constraint like `Interval(t1.val_from, t2.val_from).lengthAtLeast(Seconds(0))` is not guaranteed to be checked only after its corresponding meaningful constraint. Because of that, redundant constraints are checked more often in c5b and c5c than in c5a. Second, by construction of the queries, the intermediate results after the first step (candidate selection) are bigger in c5b and c5c than in c5a. This means that the redundant single element constraints are checked more often in these two queries than in c5a.

Overall, query complexity is a complicated concept. It has been shown that many of its aspects impact the operator's performance. In general, every query facet reducing the number of joins or the size of intermediate results will also reduce the run time. Moreover, the order in which constraints are checked is important, as demonstrated by the last experiment. This order, in turn, depends on whether the estimation component can handle the constraints.

Another aspect of queries is their selectivity.

7.3.2. Selectivity

The selectivity of a query determines its result size and is thus a crucial factor of the query processing run time as well. In order to further examine the influence of query selectivity, three variants with different selectivites are created for each of the citibike queries c5a, c5b, c5c. Now, for each of the three queries, there are variants of high, intermediate and low selectivity. Furthermore, they

are tuned so that the queries of each selectivity class have comparable result sizes. The result cardinalities for the variants are listed in Table 7.4.

	high	middle	low
c5a	162	43136	5117478
c5b	229	37247	4868340
c5c	235	47751	4832533

Table 7.4.: Result cardinalities for different versions of c5a, c5b and c5c

All nine queries are executed on citibike SF100 using 96 parallel threads. The results are given in Figure 7.6.

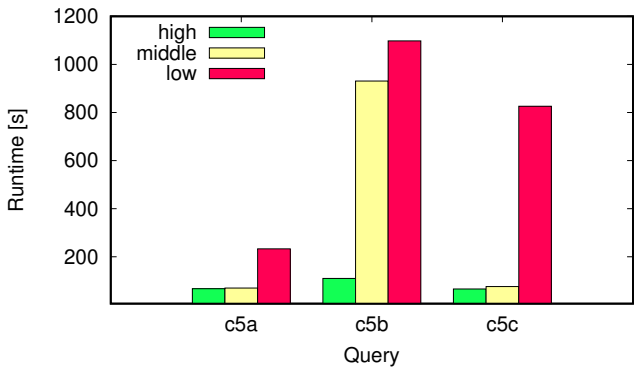

Figure 7.6.: Runtimes for different versions of c5a, c5b and c5c on citibike SF100 (parallelity 96)

As expected, higher selectivity leads to a faster processing of the query. In the queries c5a and c5c, intermediate result sizes are reduced early due to selective single element predicates. Hence, it is not surprising, that their respective version of middle selectivity is more close to the highly selective version than to the version of low selectivity.

A similar experiment is carried out for the LDBC data set. First, three variants of the query l4 are created, analogously to the citibike evaluation. The query l4a has only constraints referring to single elements. Query l4b has join predicates, but less selective single element predicates, while l4c can be seen as a compromise between l4a and l4b. For each of the three queries, three variants with high, middle and low selectivity are created. The exact queries can be found in the appendix. Their result cardinalities for SF100 are presented in Table 7.5.

	high	middle	low
l4a	807	84449	12518290
l4b	800	99895	7455989
l4c	231	73456	9241786

Table 7.5.: Result cardinalities for different versions of l5a, l5b and l5c on LDBC SF100

In Figure 7.7, the results for the LDBC SF100 data set and parallelity 96 are shown.

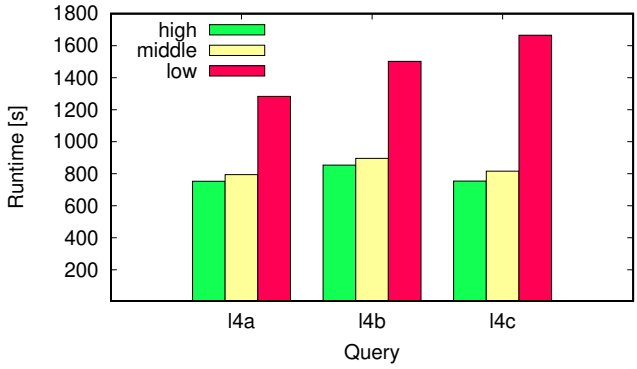

Figure 7.7.: Runtimes for different versions of l4a, l4b and l4c on LDBC 100 (parallelity 96)

The outcome resembles that of the analogous experiment for citibike data. It is clear that for each query, the highly selective version performs better than the version of middle selectivity, which in turn runs faster than the version of low selectivity. For high and middle selectivity, l4b performs the worst, as one could expect. This is different for low selectivity, which might be due to the fact that l4c has about $2 * 10^6$ results more than l4b.

For each query, the result cardinalites of high and middle selectivity and those of middle and low selectivity differ by a factor in the order of magnitude 100. Other than in the citibike evaluation, here, the run times of queries of middle selectivity are always close to those of queries of high selectivity. This indicates that intermediate results for middle and high selectivity are more similar here, which might be due to LDBC being a data set with more than one type. Hence, the labels for vertices and edges already account for many edges being filtered out during candidate selection.

In Figure 7.8, the selectivity evaluation diagrams (Figures 7.6 and 7.7) are united as one diagram.

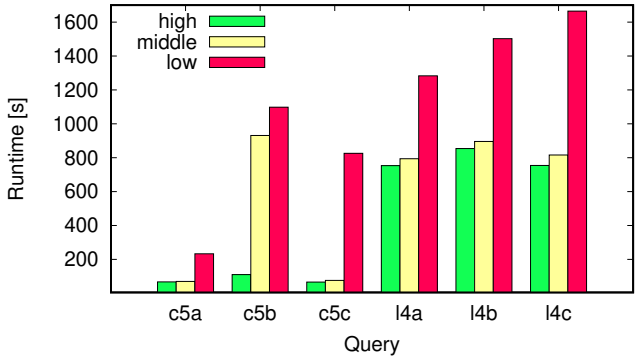

Figure 7.8.: Runtimes for different versions of c5a, c5b, c5c, l4a, l4b and l4c (parallelity 96)

Similar to the diagram in Figure 7.3, most run times of comparable queries do not mirror the two data sets' different sizes. One reason for this was already mentioned: the candidate selection of

queries on LDBC is inherently more selective because labels are employed to distinguish between different types of vertices and edges. Another reason is certainly that the result sizes of queries on citibike are quite similar to that of corresponding LDBC queries (cf. Tables 7.4, 7.5). Hence, most of the filtering is done in the first step, candidate selection. This reasoning also explains why the run time difference between the queries q5a and l4a, both only containing constraints referring to just one element, is quite notable.

This concludes the discussion of the influence of queries on the run time. Next, the operator's scalability is evaluated.

7.3.3. Scalability

Gradoop is capable of processing large data sets in parallel. Thus, it is of high interest how the operator performs with respect to different data set sizes and degrees of parallelity.

A representative set of queries is selected and evaluated on the three citibike data sets using parallelity 96 (16 workers). The results are visualized in Figure 7.9, now with logarithmic y-axis as the data set sizes differ by factor 10.

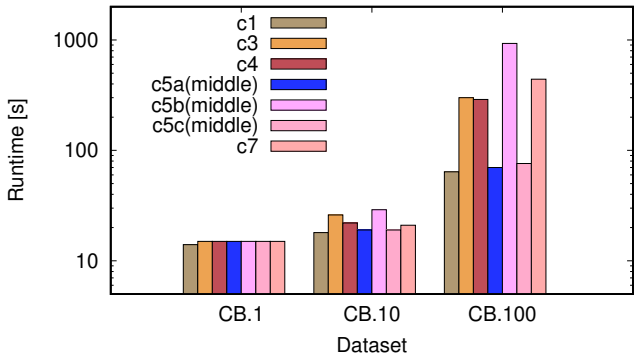

Figure 7.9.: Runtimes for different queries on citibike SF1, SF10 and SF100 (parallelity 96)

The run times for citibike SF1 hardly differ at all, implying that on such small data sets, a wide range of different queries can be processed in similar time. Here, all run times are about 15 seconds. For SF10, run time differences for different queries surface. These differences are analogous to the run time differences in SF100, however, they are less extreme with run times ranging from 18 seconds (c1) to 29 seconds (c5b). Furthermore, the run times for SF10 do not even double the run times for SF1. Hence, the operator scales good for comparatively small data sets, the speedup is superlinear here. The results for SF100 do not scale that well: while the run time for c1 is 64 seconds here (18s for SF10), c5b now takes 931 seconds to process (29s for SF10).

So, the scalability in terms of data set sizes depends on query complexities for larger data sets, while for small data sets like SF1 and SF10, the operator scales superlinearly for various kinds of queries - at least for a sufficiently high parallelity.

Scalability is also evaluated on the LDBC data set. Figure 7.11 displays the run times of the queries l0, l1,...,l7 for all three data sets.

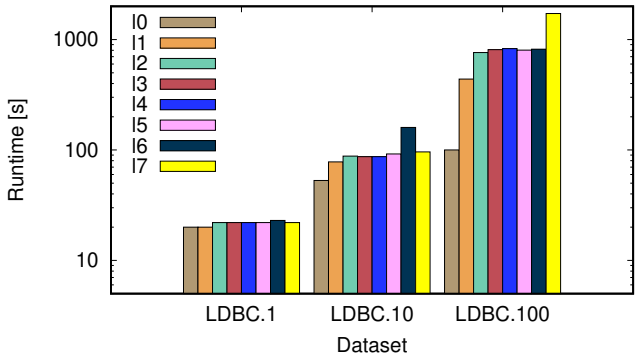

Figure 7.10.: Runtimes for different queries on LDBC SF1, SF10 and SF100 (parallelity 96)

The speedup from SF1 compared to SF10 is superlinear in all cases, too. In the worst case, the run time for SF10 differs from that for SF10 by a factor of 7 (l6). Similarly, the speedup from SF10 compared to SF100 is superlinear in almost every case. There is only one exception, as l6 takes 160 seconds for SF10 but 1721 seconds for SF100 (factor 10.76).

Another experiment with the variants of the queries l4a, l4b and l4c yields similar results, as shown in Figure 7.11.

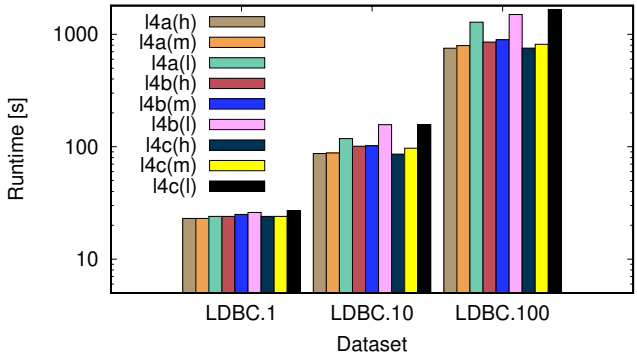

Figure 7.11.: Runtimes for different variations of l4 on LDBC SF1, SF10 and SF100 (parallelity 96)

Again, the speedup from SF1 to SF10 is superlinear for every query. This is also true for the speedup from SF10 to SF100, except for the queries l4b (high selectivity) and l4c (low selectivity). However, for these two queries, the speedup is almost linear, too. Overall, the run time scales satisfactorily for the bigger data set, LDBC.

Parallelity is another aspect of scalability. The operator is thus evaluated for different numbers of workers: 1,2,4,8 and 16 workers corresponding to 6,12,24,48 and 96 parallel threads. First, parallelity

is investigated for the citibike data set. Again, a set of queries from the ones used before is selected. Now, however, queries that were processed comparatively fast are preferred in order not to provoke too high run times for small degrees of parallelity. Figure 7.12 shows the results.

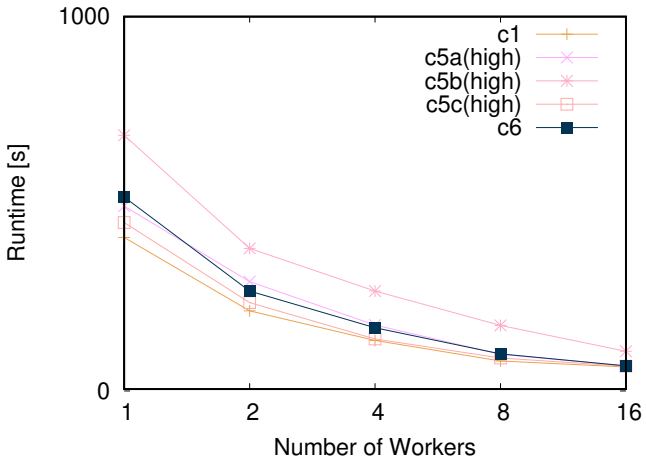

Figure 7.12.: Runtimes of citibike queries for different degrees of parallelity (citibike SF100)

The curves are of similar shape for all tested queries, indicating that the speedup for different degrees of parallelity is not heavily dependent on the type of query. In Figure 7.13, the speedup curves are plotted.

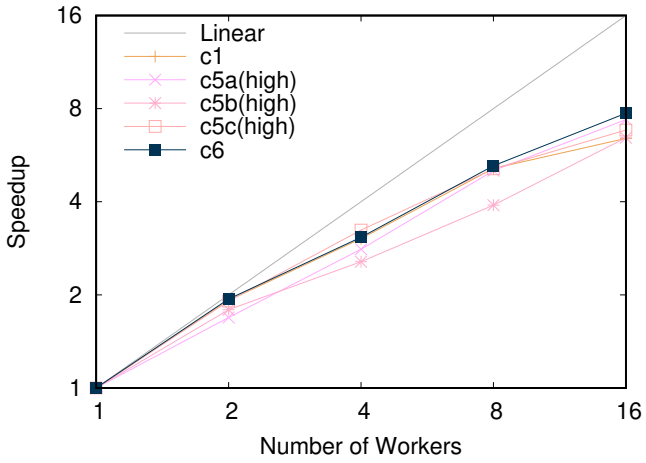

Figure 7.13.: Speedup of citibike queries for different degrees of parallelity (citibike SF100)

Here it becomes clear, that the speedup for the different queries is indeed very similar. There is only one exception: the speedup of query c5b is significantly worse than that of the other queries, at least for 4 and 8 workers. For the other queries, the speedup is nearly linear for up to 4 workers. It is not surprising that the speedup drops for an increasing number of workers, as more workers also lead to more communication effort between workers. Additionally, the citibike data set is rather small so that higher parallelities do not have a notable effect on query processing.

The scalability with respect to different degrees of parallelity is evaluated for LDBC data, too. Figure 7.14 plots the run times of a diverse set of queries, again for 1, 2, 4, 8 and 16 workers, corresponding to 6, 12, 24, 48 and 96 parallel threads.

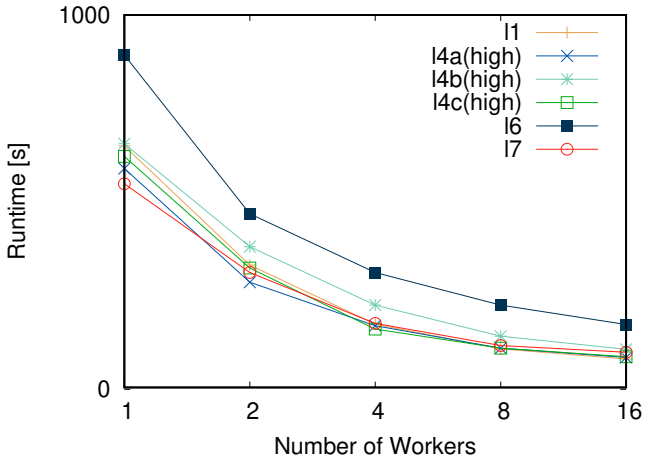

Figure 7.14.: Runtimes of LDBC queries for different degrees of parallelity (LDBC SF100)

All queries behave similarly: there is considerable speedup for 1 to 4 workers, while using 16 workers does not significantly improve the runtime compared to the use of 8 workers. This, again, may be due to higher communication costs for more workers. In Figure 7.15, the speed up for these run times is visualized.

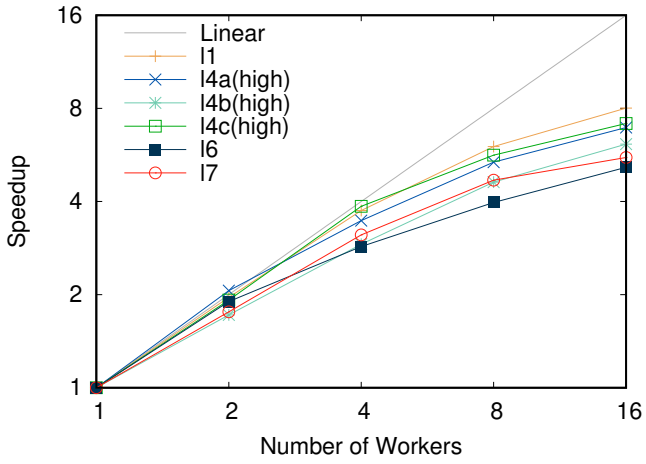

Figure 7.15.: Speedup of LDBC queries for different degrees of parallelity (LDBC SF100)

For up to 4 workers, it is nearly linear, even superlinear in one case, namely l4a(high) for 2 workers. Junghanns et al. [28] also report super-linear speedups. They argue that this is because for less main memory, intermediate results are written to disk. For more main memory, however, no such swapping is necessary.

Figure 7.16 combines speedup curves from Figures 7.13 and 7.15 for some comparable queries on the citibike and the LDBC data sets.

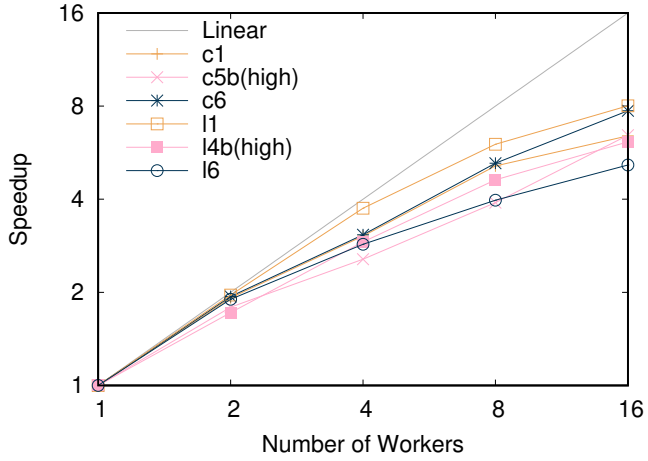

Figure 7.16.: Speedup of citibike and LDBC queries for different degrees of parallelity (citibike SF100, LDBC SF100)

It does not allow to formulate hypotheses on how the speed up depends on the type of data graph. The curves are, in general, very similar. There is no pattern of similar queries behaving differently on different data sets.

7.3.4. Comparison between EPGM and TPGM Pattern Matching

Besides measuring run times for different queries and data graphs, it is interesting how the TPGM operator's performance compares to that of the EPGM operator. However, investigating this question is beyond the scope of this thesis. Comparing the two operators is not trivial.

If one were to evaluate both operators, only non-temporal queries could be considered, as the EPGM implementation is not able to process temporal GDL queries and TPGM graphs.

Another problem is that most optimizations implemented in this thesis focus on temporal queries. Hence, for non-temporal queries, the only difference between the two operators are the property values' estimations (Sections 6.2.5 and 6.2.6) in the TPGM operator.

Junghanns et al. [28] already evaluated the EPGM pattern matching operator. Unfortunately, the results can not be compared to the results obtained here. In [28], another data format was used to store data and write results to. Moreover, many of Junghanns et al.'s [28] queries contained path expressions. The queries used for evaluation here do not consider them, as they were not part of the TPGM pattern matching implementation. The results obtained by Junghanns et al. [28] show surprisingly fast run times. These run times, however, could not be reproduced using the evaluation set up described in Section 7.1. Instead, the processing of the queries used by Junghanns et al., in most cases, took significantly longer than processing the queries used here.

8. Discussion

In this thesis, temporal pattern matching in Gradoop has been implemented. Gradoop's query language GDL was extended with several temporal data types and predicates. The existing EPGM pattern matching implementation was modified to enable Gradoop to process temporal queries. Moreover, two attempts at speeding up the operator's run time were made: estimations for predicate selectivities were provided and a query rewriting pipeline constructed.

All of these aspects leave room for improvement in future work.

Temporal predicates in GDL could be more expressive and flexible. For example, different time zones can not be handled as of now. Furthermore, there are only 3 options to create a constant time stamp, see Table 4.1. Another important aspect neglected in this thesis are temporal path expressions, as pointed out in Section 4.7.

The query pipeline introduced in Section 6.1 may be improved, too. The inference transformation (Section 6.1.7) only considers singleton temporal CNF clauses and could thus be refined to take more aspects into consideration. Temporal Subsumption (Section 6.1.8) is rather simplistic. Here, more sublime implications could be implemented. What is more, the pipeline ignores duration and non-temporal property constraints.

Section 6.2 presented estimations for property values and structural properties. It also pointed out the shortcomings of the chosen approach. Especially, the estimation of structural properties, e.g. the number of distinct source/target vertices for a given edge, could be optimized. Other sampling approaches, particularly structure-preserving ones, can be implemented and compared to the provided Reservoir Sampling estimations. Also, more sophisticated methods to approximate the distributions of property values might be conceived.

Bibliography

[1] Jans Aasman. Allegro graph: Rdf triple database. *Cidade: Oakland Franz Incorporated*, 17, 2006.

[2] James F Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[3] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 44–54, 2006.

[4] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1199–1214, 2016.

[5] Jaewook Byun, Sungpil Woo, and Daeyoung Kim. Chronograph: Enabling temporal graph traversals for efficient information diffusion analysis over time. *IEEE Transactions on Knowledge and Data Engineering*, 32(3):424–437, 2019.

[6] Vincenzo Carletti, Pasquale Foggia, Alessia Saggese, and Mario Vento. Introducing vf3: A new algorithm for subgraph isomorphism. In *International Workshop on Graph-Based Representations in Pattern Recognition*, pages 128–139. Springer, 2017.

[7] Xiaoying Chen, Chong Zhang, Bin Ge, and Weidong Xiao. Temporal social network: Storage, indexing and query processing. In *EDBT/ICDT Workshops*, 2016.

[8] James Cheng, Yiping Ke, Wilfred Ng, and An Lu. Fg-index: towards verification-free query processing on graph databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 857–872, 2007.

[9] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 85–98, 2012.

[10] World Wide Web Consortium et al. Rdf 1.1 concepts and abstract syntax. 2014.

[11] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *International journal of pattern recognition and artificial intelligence*, 18(03):265–298, 2004.

[12] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

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

[14] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. Graph pattern matching: from intractable to polynomial time. *Proceedings of the VLDB Endowment*, 3(1-2):264–275, 2010.

[15] Arash Fard, M Usman Nisar, Lakshmish Ramaswamy, John A Miller, and Matthew Saltz. A distributed vertex-centric approach for pattern matching in massive graphs. In *2013 IEEE International Conference on Big Data*, pages 403–411. IEEE, 2013.

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

[17] Maximilian Franzke, Tobias Emrich, Andreas Züfle, and Matthias Renz. Pattern search in temporal social networks. In *Proceedings of the 21st International Conference on Extending Database Technology*, 2018.

[18] Swapnil Gandhi and Yogesh Simmhan. An interval-centric model for distributed computing over temporal graphs. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1129–1140. IEEE, 2020.

[19] Hector Garcia-Molina, Jeffrey D Ullman, and Jennifer Widom. *Database system implementation*, volume 672. Prentice Hall Upper Saddle River, NJ:, 2000.

[20] Kevin Gomez, Matthias Täschner, M Ali Rostami, Christopher Rost, and Erhard Rahm. Graph sampling with distributed in-memory dataflow systems. *arXiv preprint arXiv:1910.04493*, 2019.

[21] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[22] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 337–348, 2013.

[23] Steve Harris, Andy Seaborne, and E Prudâhommeaux. Sparql 1.1 query language. w3c recommendation (2013). *URL https://www. w3. org/TR/sparql11-query*, 2013.

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

[25] Petter Holme and Jari Saramäki. Temporal networks. *Physics reports*, 519(3):97–125, 2012.

[26] Haixing Huang, Jinghe Song, Xuelian Lin, Shuai Ma, and Jinpeng Huai. Tgraph: A temporal graph data management system. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 2469–2472, 2016.

[27] Silu Huang, James Cheng, and Huanhuan Wu. Temporal graph traversals: Definitions, algorithms, and applications. *arXiv preprint arXiv:1401.1919*, 2014.

[28] Martin Junghanns, Max Kießling, Alex Averbuch, André Petermann, and Erhard Rahm. Cypher-based graph pattern matching in gradoop. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, pages 1–8, 2017.

[29] Martin Junghanns, Max Kießling, Niklas Teichmann, Kevin Gómez, André Petermann, and Erhard Rahm. Declarative and distributed graph analytics with gradoop. *Proceedings of the VLDB Endowment*, 11(12):2006–2009, 2018.

[30] Martin Junghanns, André Petermann, Martin Neumann, and Erhard Rahm. Management and analysis of big graph data: current systems and open challenges. In *Handbook of Big Data Technologies*, pages 457–505. Springer, 2017.

[31] Martin Junghanns, André Petermann, and Erhard Rahm. Distributed grouping of property graphs with gradoop. *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*, 2017.

[32] Martin Junghanns, André Petermann, Niklas Teichmann, Kevin Gómez, and Erhard Rahm. Analyzing extended property graphs with apache flink. In *Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics*, pages 1–8, 2016.

[33] Vassilis Kostakos. Temporal graphs. *Physica A: Statistical Mechanics and its Applications*, 388(6):1007–1023, 2009.

[34] Lauri Kovanen, Márton Karsai, Kimmo Kaski, János Kertész, and Jari Saramäki. Temporal motifs in time-dependent networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2011(11):P11005, 2011.

[35] Matthias Kricke, Eric Peukert, and Erhard Rahm. Graph data transformations in gradoop. *BTW 2019*, 2019.

[36] Krishna Kulkarni and Jan-Eike Michels. Temporal features in sql: 2011. *ACM Sigmod Record*, 41(3):34–43, 2012.

[37] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *Proceedings of the VLDB Endowment*, 6(2):133–144, 2012.

[38] Jure Leskovec and Christos Faloutsos. Sampling from large graphs. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 631–636, 2006.

[39] Youhuan Li, Lei Zou, M Tamer Özsu, and Dongyan Zhao. Time constrained continuous subgraph search over streaming graphs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1082–1093. IEEE, 2019.

[40] Wouter Lightenberg, Yulong Pei, George Fletcher, and Mykola Pechenizkiy. Tink: A temporal graph analytics library for apache flink. In *Companion Proceedings of the The Web Conference 2018*, pages 71–72, 2018.

[41] Ziyang Liu, Chong Wang, and Yi Chen. Keyword search on temporal graphs. *IEEE Transactions on Knowledge and Data Engineering*, 29(8):1667–1680, 2017.

[42] Shuai Ma, Yang Cao, Wenfei Fan, Jinpeng Huai, and Tianyu Wo. Capturing topology in graph pattern matching. *arXiv preprint arXiv:1201.0229*, 2011.

[43] Shuai Ma, Yang Cao, Wenfei Fan, Jinpeng Huai, and Tianyu Wo. Strong simulation: Capturing topology in graph pattern matching. *ACM Transactions on Database Systems (TODS)*, 39(1):1–46, 2014.

[44] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[45] Norbert Martinez-Bazan, Sergio Gomez-Villamor, and Francesc Escale-Claveras. Dex: A high-performance graph database management system. In *2011 IEEE 27th International Conference on Data Engineering Workshops*, pages 124–127. IEEE, 2011.

[46] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. Immortalgraph: A system for storage and analysis of temporal graphs. *ACM Transactions on Storage (TOS)*, 11(3):1–34, 2015.

[47] Othon Michail. An introduction to temporal graphs: An algorithmic perspective. *Internet Mathematics*, 12(4):239–280, 2016.

[48] Justin J Miller. Graph database applications and concepts with neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, volume 2324, 2013.

[49] Ashwin Paranjape, Austin R Benson, and Jure Leskovec. Motifs in temporal networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 601–610, 2017.

[50] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[51] André Petermann and Martin Junghanns. Scalable business intelligence with graph collections. *it Inf. Technol.*, 58(4):166–175, 2016.

[52] André Petermann, Martin Junghanns, Stephan Kemper, Kevin Gómez, Niklas Teichmann, and Erhard Rahm. Graph mining for complex data analytics. In *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*, pages 1316–1319. IEEE, 2016.

[53] André Petermann, Martin Junghanns, and Erhard Rahm. Dimspan: Transactional frequent subgraph mining with distributed in-memory dataflow systems. In *Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*, pages 237–246, 2017.

[54] Ursula Redmond and Pádraig Cunningham. Temporal subgraph isomorphism. In *2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2013)*, pages 1451–1452. IEEE, 2013.

[55] Xuguang Ren and Junhu Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proceedings of the VLDB Endowment*, 8(5):617–628, 2015.

[56] Christopher Rost, Andreas Thor, Philip Fritzsche, Kevin Gomez, and Erhard Rahm. Evolution analysis of large graphs with gradoop. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 402–408. Springer, 2019.

[57] Christopher Rost, Andreas Thor, and Erhard Rahm. Temporal graph analysis using gradoop. *BTW 2019–Workshopband*, 2019.

[58] M Ali Rostami, Matthias Kricke, Eric Peukert, Stefan Kühne, Moritz Wilke, Steffen Dienst, and Erhard Rahm. Biggr: Bringing gradoop to applications. *Datenbank-Spektrum*, 19(1):51–60, 2019.

[59] Polina Rozenshtein and Aristides Gionis. Temporal pagerank. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 674–689. Springer, 2016.

[60] Matthew Saltz, Ayushi Jain, Abhishek Kothari, Arash Fard, John A Miller, and Lakshmish Ramaswamy. Dualiso: An algorithm for subgraph pattern matching on very large labeled graphs. In *2014 IEEE International Congress on Big Data*, pages 498–505. IEEE, 2014.

[61] Konstantinos Semertzidis and Evaggelia Pitoura. Top-k durable graph pattern queries on temporal graphs. *IEEE Transactions on Knowledge and Data Engineering*, 31(1):181–194, 2018.

[62] Konstantinos Semertzidis and Evaggelia Pitoura. A hybrid approach to temporal pattern matching. *arXiv preprint arXiv:2001.01661*, 2020.

[63] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proceedings of the VLDB Endowment*, 1(1):364–375, 2008.

[64] Chunyao Song, Tingjian Ge, Cindy Chen, and Jie Wang. Event pattern matching over graph streams. *Proceedings of the VLDB Endowment*, 8(4):413–424, 2014.

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

[66] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. Pgql: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, pages 1–6, 2016.

[67] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[68] Charith Wickramaarachchi, Rajgopal Kannan, Charalampos Chelmis, and Viktor K Prasanna. Distributed exact subgraph matching in small diameter dynamic graphs. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 3360–3369. IEEE, 2016.

[69] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. Path problems in temporal graphs. *Proceedings of the VLDB Endowment*, 7(9):721–732, 2014.

[70] Huanhuan Wu, Yuzhen Huang, James Cheng, Jinfeng Li, and Yiping Ke. Reachability and time-based path queries in temporal graphs. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 145–156. IEEE, 2016.

[71] Xifeng Yan, Philip S Yu, and Jiawei Han. Graph indexing: a frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 335–346, 2004.

[72] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. Triplebit: a fast and compact system for large scale rdf data. *Proceedings of the VLDB Endowment*, 6(7):517–528, 2013.

[73] Shijie Zhang, Shirong Li, and Jiong Yang. Gaddi: distance index based subgraph matching in biological networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 192–203, 2009.

[74] Peixiang Zhao and Jiawei Han. On graph query optimization in large networks. *Proceedings of the VLDB Endowment*, 3(1-2):340–351, 2010.

A. Evaluation Data

The following lists all queries used in the evaluation and their measured run times.

A. LDBC Data

A.1. Queries

```
1  MATCH  (p1:person)
```

Listing A.1: Query l0

```
1  MATCH  (p1:person)-[l:likes]->(m)
2  WHERE  l.tx_from >= 2012-06-01 AND
3         l.tx_from <= 2012-06-01T23:59:59 AND
4         m.tx_from >=2012-05-30 AND
5         m.tx_from <= 2012-06-01T23:59:59
```

Listing A.2: Query l1

```
1  MATCH  (p1:person)-[l1:likes] ->(m)<-[l2:likes]-(p2:person)
2  WHERE  l1.tx_from >= 2012-06-01 AND
3         l2.tx_from >= 2012-06-01 AND
4         l1.tx_from <= 2012-06-01T23:59:59 AND
5         l2.tx_from <= 2012-06-01T23:59:59 AND
6         p1.tx_to > 1970-01-01 AND
7         m.tx_from >= 2012-05-30 AND
8         m.tx_from <= 2012-06-01T23:59:59
```

Listing A.3: Query l2

```
1  MATCH  (p:person)-[l:likes]->(c:comment)-[r:replyOf]->(post:post)
2  WHERE  l.tx_from >= 2012-06-01 AND
3         l.tx_from <= 2012-06-01T23:59:59 AND
4         p.tx_to > 1970-01-01 AND
5         c.tx_from >= 2012-05-30 AND
6         c.tx_from <= 2012-06-01T23:59:59 AND
7         post.tx_from >= 2012-05-30 AND
8         post.tx_from <= 2012-06-01T23:59:59
```

Listing A.4: Query l3

```
1  MATCH  (p1:person)-[k:knows]->(p2:person)
2         (p2)-[l1:likes]->(m)<-[l2:likes]-(p1)
3  WHERE  l1.tx_from >= 2012-06-01 AND
4         l2.tx_from >= 2012-06-01 AND
5         l1.tx_from <= 2012-06-01T23:59:59 AND
6         l2.tx_from <= 2012-06-01T23:59:59 AND
7         p1.tx_to > 1970-01-01 AND
```

```
8         m.tx_from >= 2012-05-30 AND
9         m.tx_from <= 2012-06-01T23:59:59
```

Listing A.5: Query l4

```
1  MATCH  (p1:person)-[l1:likes]->(c:comment)
2         (c)-[r:replyOf]->(m)<-[l2:likes]-(p1)
3  WHERE  l1.tx_from >= 2012-06-01 AND
4         l2.tx_from >= 2012-06-01 AND
5         l1.tx_from <= 2012-06-01T23:59:59 AND
6         l2.tx_from <= 2012-06-01T23:59:59 AND
7         p1.tx_to > 1970-01-01 AND
8         m.tx_from >= 2012-05-30 AND
9         m.tx_from <= 2012-06-01T23:59:59 AND
10        c.tx_from >= 2012-05-30 AND
11        c.tx_from <= 2012-06-01T23:59:59
```

Listing A.6: Query l5

```
1  MATCH  (p1:person)-[k1:knows]->(p2:person)
2         (p2)-[l1:likes]->(m)<-[l2:likes]-(p3:person)
3         (p3)<-[k2:knows]-(p1)
4  WHERE  l1.tx_from >= 2012-06-01 AND
5         l2.tx_from >= 2012-06-01 AND
6         l1.tx_from <=2012-06-01T23:59:59 AND
7         l2.tx_from <= 2012-06-01T23:59:59 AND
8         p1.tx_to > 1970-01-01 AND
9         m.tx_from >= 2012-05-30 AND
10        m.tx_from <= 2012-06-01T23:59:59
```

Listing A.7: Query l6

```
1  MATCH  (p1:person)-[l1:likes]->(c1)
2         (c1)-[r1:replyOf]->(m)<-[r2:replyOf]-(c2)
3         (c2)<-[l2:likes]-(p1)
4  WHERE  l1.tx_from >= 2012-06-01 AND
5         l2.tx_from >= 2012-06-01 AND
6         l1.tx_from <= 2012-06-01T23:59:59 AND
7         l2.tx_from <= 2012-06-01T23:59:59 AND
8         p1.tx_to > 1970-01-01 AND
9         m.tx_from >= 2012-05-30 AND
10        m.tx_from <= 2012-06-01T23:59:59 AND
11        c1.tx_from >= 2012-05-30 AND
12        c1.tx_from <= 2012-06-01T23:59:59 AND
13        c2.tx_from >= 2012-05-30 AND
14        c2.tx_from <= 2012-06-01T23:59:59
```

Listing A.8: Query l7

```
1  MATCH  (p1:person)-[k:knows]->(p2:person)
2         (p2)-[l1:likes]->(m)<-[l2:likes]-(p1)
```

```
3  WHERE  l1.tx_from >= 2012-06-01 AND
4         l2.tx_from >= 2012-06-01 AND
5         l1.tx_from <= 2012-06-07 AND
6         l2.tx_from <= 2012-06-07 AND
7         p1.tx_to > 1970-01-01 AND
8         m.tx_from >= 2012-06-01 AND
9         m.tx_from <= 2012-06-07
```

Listing A.9: Query l4a (low)

```
1  MATCH  (p1:person)-[k:knows]->(p2:person)
2         (p2)-[l1:likes]->(m)<-[l2:likes]-(p1)
3  WHERE  l1.tx_from >= 2012-06-01 AND
4         l2.tx_from >= 2012-06-01 AND
5         l1.tx_from <= 2012-06-07 AND
6         l2.tx_from <= 2012-06-07 AND
7         p1.tx_to > 1970-01-01 AND
8         m.tx_from >= 2012-06-01 AND
9         m.tx_from < 2012-06-02 AND
10        p1.browserUsed="Chrome" AND
11        p2.browserUsed="Chrome"
```

Listing A.10: Query l4a (middle)

```
1  MATCH  (p1:person)-[k:knows]->(p2:person)
2         (p2)-[l1:likes]->(m)<-[l2:likes]-(p1)
3  WHERE  l1.tx_from >= 2012-06-01 AND
4         l2.tx_from >= 2012-06-01 AND
5         l1.tx_from <= 2012-06-02 AND
6         l2.tx_from <= 2012-06-02 AND
7         p1.tx_to > 1970-01-01 AND
8         m.tx_from >= 2012-06-01 AND
9         m.tx_from < 2012-06-02 AND
10        p1.browserUsed="Safari" AND
11        p2.browserUsed="Chrome"
```

Listing A.11: Query l4a (high)

```
1  MATCH  (p1:person)-[k:knows]->(p2:person)
2         (p2)-[l1:likes]->(m)<-[l2:likes]-(p1)
3  WHERE  l1.tx_from >= 2012-06-01 AND
4         l2.tx_from >= 2012-06-01 AND
5         l1.tx_from <= 2012-06-28 AND
6         l2.tx_from <= 2012-06-28 AND
7         p1.tx_to > 1970-01-01 AND
8         m.tx_from >= 2012-06-01 AND
9         m.tx_from < 2012-06-28 AND
10        Interval(l1.tx_from, l2.tx_from).lengthAtMost(Hours(36)) AND
11        p1.gender=p2.gender
```

Listing A.12: Query l4b (low)

```
MATCH  (p1:person)-[k:knows]->(p2:person)
       (p2)-[l1:likes]->(m)<-[l2:likes]-(p1)
WHERE  l1.tx_from >= 2012-06-01 AND
       l2.tx_from >= 2012-06-01 AND
       l1.tx_from <= 2012-06-28 AND
       l2.tx_from <= 2012-06-28 AND
       p1.tx_to > 1970-01-01 AND
       m.tx_from >= 2012-06-01 AND
       m.tx_from < 2012-06-28 AND
       Interval(l1.tx_from, l2.tx_from).lengthAtMost(Hours(1)) AND
       p1.gender=p2.gender AND
       p1.browserUsed=p2.browserUsed
```

Listing A.13: Query l4b (middle)

```
MATCH  (p1:person)-[k:knows]->(p2:person)
       (p2)-[l1:likes]->(m)<-[l2:likes]-(p1)
WHERE  l1.tx_from >= 2012-06-01 AND
       l2.tx_from >= 2012-06-01 AND
       l1.tx_from <= 2012-06-28 AND
       l2.tx_from <= 2012-06-28 AND
       p1.tx_to > 1970-01-01 AND
       m.tx_from >= 2012-06-01 AND
       m.tx_from < 2012-06-28 AND
       Interval(l1.tx_from, l2.tx_from).lengthAtMost(Seconds(30)) AND
       p1.gender=p2.gender AND
       p1.browserUsed=p2.browserUsed AND
       l1.tx_from < l2.tx_from
```

Listing A.14: Query l4b (high)

```
MATCH  (p1:person)-[k:knows]->(p2:person)
       (p2)-[l1:likes]->(m)<-[l2:likes]-(p1)
WHERE  l1.tx_from >= 2012-06-01 AND
       l2.tx_from >= 2012-06-01 AND
       l1.tx_from <= 2012-06-21 AND
       l2.tx_from <= 2012-06-21 AND
       p1.tx_to > 1970-01-01 AND
       m.tx_from >= 2012-06-01 AND
       m.tx_from < 2012-06-21 AND
       Interval(l1.tx_from, l2.tx_from).lengthAtMost(Hours(96)) AND
       p1.gender=p2.gender
```

Listing A.15: Query l4c (low)

```
MATCH  (p1:person)-[k:knows]->(p2:person)
       (p2)-[l1:likes]->(m)<-[l2:likes]-(p1)
WHERE  l1.tx_from >= 2012-06-01 AND
       l2.tx_from >= 2012-06-01 AND
       l1.tx_from <= 2012-06-10 AND
```

```
6           l2.tx_from <= 2012-06-10 AND
7           p1.tx_to > 1970-01-01 AND
8           m.tx_from >= 2012-06-01 AND
9           m.tx_from < 2012-06-10 AND
10          Interval(l1.tx_from, l2.tx_from).lengthAtMost(Hours(2)) AND
11          p1.gender=p2.gender AND
12          p1.browserUsed=p2.browserUsed
```

Listing A.16: Query l4c (middle)

```
1   MATCH   (p1:person)-[k:knows]->(p2:person)
2           (p2)-[l1:likes]->(m)<-[l2:likes]-(p1)
3   WHERE   l1.tx_from >= 2012-06-01 AND
4           l2.tx_from >= 2012-06-01 AND
5           l1.tx_from <= 2012-06-04 AND
6           l2.tx_from <= 2012-06-04 AND
7           p1.tx_to > 1970-01-01 AND
8           m.tx_from >= 2012-06-01 AND
9           m.tx_from < 2012-06-03 AND
10          Interval(l1.tx_from, l2.tx_from).lengthAtMost(Minutes(2)) AND
11          p1.gender=p2.gender AND
12          p1.browserUsed=p2.browserUsed
```

Listing A.17: Query l4c (high)

A.2. Cardinalities and Runtimes

	SF1	SF10	SF100
l0	9892	65645	448626
l1	999	17353	211945
l2	17262	1076784	12582910
l3	353	5583	76301
l4	558	14059	222350
l5	9	47	608
l6	24880	1196184	22141686
l7	2	10	622
l4a (l)	54355	745551	12518290
l4a (m)	242	5448	84449
l4a (h)	0	39	807
l4b (l)	28781	438082	7455989
l4b (m)	364	5794	99895
l4b (h)	0	50	800
l4c (l)	36884	542353	9241786
l4c (m)	334	4428	73456
l4c (h)	1	12	231

Table A.1.: Cardinalities of LDBC queries

	SF1	SF10	SF100
l0	20	53	439
l1	20	78	763
l2	22	88	811
l3	22	87	828
l4	22	87	802
l5	22	92	818
l6	23	160	1721
l7	22	96	870
l4a (l)	24	118	1283
l4a (m)	23	88	794
l4a (h)	23	87	753
l4b (l)	26	151	1502
l4b (m)	25	102	896
l4b (h)	24	101	854
l4c (l)	27	157	1665
l4c (m)	24	97	816
l4c (h)	24	86	754

Table A.2.: Runtimes of LDBC queries (Parallelity 96)

	6	12	24	48	96
l1	650 (1.)	332 (1.96)	174 (3.74)	108 (6.02)	81 (8.02)
l6	893 (1.)	469 (1.90)	312 (2.86)	225 (3.97)	173 (5.16)
l7	549 (1.)	312 (1.76)	176 (3.12)	117 (4.69)	99 (5.55)
l4a (h)	590 (1.)	286 (2.06)	170 (3.47)	110 (5.36)	85 (6.94)
l4b (h)	656 (1.)	381 (1.72)	225 (2.91)	142 (4.62)	107 (6.13)
l4c (h)	622 (1.)	324 (1.92)	161 (3.86)	110 (5.65)	87 (7.15)

Table A.3.: Runtimes and speedup of LDBC queries for different degrees of parallelity (SF10)

B. Citibike Data

B.1. Queries

```
1  MATCH  (s1)
```

Listing A.18: Query c0

```
1  MATCH  (s1)-[t]->(s2)
2  WHERE  2017-08-01 <= t.val_from AND
3         t.val_from <= 2017-08-01T23:59:59 AND
4         s1.tx_to>2020-01-01
```

Listing A.19: Query c1

```
1  MATCH  (s1)-[t1]->(s1)
2  WHERE  2017-08-01 <= t1.val_from AND
3         t1.val_from <= 2017-08-01T23:59:59 AND
```

```
4         s1.tx_to>2020-01-01
```

Listing A.20: Query c2

```
1  MATCH  (s1)-[t1]->(s2)<-[t2]-(s3)
2  WHERE  2017-08-01 <= t1.val_from AND
3         t1.val_from <= 2017-08-01T23:59:59 AND
4         2017-08-01 <= t2.val_from AND
5         t2.val_from <= 2017-08-01T23:59:59 AND
6         s1.tx_to>2020-01-01
```

Listing A.21: Query c3

```
1  MATCH  (s1)-[t1]->(s2)-[t2]->(s3)
2  WHERE  2017-08-01 <= t1.val_from AND
3         t1.val_from <= 2017-08-01T23:59:59 AND
4         2017-08-01 <= t2.val_from AND
5         t2.val_from <= 2017-08-01T23:59:59 AND
6         s1.tx_to>2020-01-01|
```

Listing A.22: Query c4

```
1  MATCH  (s3)<-[t2]-(s1)-[t1]->(s2)
2  WHERE  2017-08-01 <= t1.val_from AND
3         t1.val_from <= 2017-08-01T23:59:59 AND
4         2017-08-01 <= t2.val_from AND
5         t2.val_from <= 2017-08-01T23:59:59 AND
6         s1.tx_to>2020-01-01
```

Listing A.23: Query c5

```
1  MATCH  (s1)-[t1]->(s2)-[t2]->(s3)<-[t3]-(s1)
2  WHERE  2017-08-01T12:00:00 <= t1.val_from AND
3         t1.val_from <= 2017-08-01T13:00:00 AND
4         2017-08-01T12:00:00 <= t2.val_from AND
5         t2.val_from <= 2017-08-01T13:00:00 AND
6         2017-08-01T12:00:00 <= t3.val_from AND
7         t3.val_from <= 2017-08-01T13:00:00 AND
8         s1.tx_to>2020-01-01
```

Listing A.24: Query c6

```
1  MATCH  (s1)-[t1]->(s2)-[t2]->(s3)<-[t3]-(s4)
2  WHERE  2017-08-01T17:00:00 <= t1.val_from AND
3         t1.val_from <= 2017-08-01T18:30:00 AND
4         2017-08-01T17:00:00 <= t2.val_from AND
5         t2.val_from <= 2017-08-01T18:30:00 AND
6         2017-08-01T17:00:00 <= t3.val_from AND
7         t3.val_from <= 2017-08-01T18:30:00 AND
8         s1.tx_to>2020-01-01
```

Listing A.25: Query c7

```
MATCH  (s3)<-[t2]-(s1)-[t1]->(s2)
WHERE  2017-08-01 <= t1.val_from AND
       t1.val_from <= 2017-08-08 AND
       2017-08-01 <= t2.val_from AND
       t2.val_from <= 2017-08-08 AND
       s1.capacity >= 50 AND
       s2.capacity <= 25 AND
       s3.capacity <= 25 AND
       s1.tx_to>2020-01-01
```

Listing A.26: Query c5a (low)

```
MATCH  (s3)<-[t2]-(s1)-[t1]->(s2)
WHERE  2017-08-01 <= t1.val_from AND
       t1.val_from <= 2017-08-02 AND
       2017-08-01 <= t2.val_from AND
       t2.val_from <= 2017-08-02 AND
       s1.capacity >= 68 AND
       s2.capacity <= 20 AND
       s3.capacity <= 20 AND
       s1.tx_to>2020-01-01
```

Listing A.27: Query c5a (middle)

```
MATCH  (s3)<-[t2]-(s1)-[t1]->(s2)
WHERE  2017-08-01T09:00:00 <= t1.val_from AND
       t1.val_from <= 2017-08-01T12:00:00 AND
       2017-08-01T09:00:00 <= t2.val_from AND
       t2.val_from <= 2017-08-01T12:00:00 AND
       s1.capacity >= 75 AND
       s2.capacity <= 18 AND
       s3.capacity <= 18 AND
       s1.tx_to>2020-01-01
```

Listing A.28: Query c5a (high)

```
MATCH  (s3)<-[t2]-(s1)-[t1]->(s2)
WHERE  2017-08-01 <= t1.val_from AND
       t1.val_from <= 2017-09-01 AND
       2017-08-01 <= t2.val_from AND
       t2.val_from <= 2017-09-01 AND
       t2.val_from > t1.val_from AND
       Interval(t1.val_from, t2.val_from).lengthAtMost(Minutes(30)) AND
       s1.capacity >= s2.capacity AND
       s1.capacity >= s3.capacity AND
       s1.tx_to>2020-01-01
```

Listing A.29: Query c5b (low)

```
MATCH  (s3)<-[t2]-(s1)-[t1]->(s2)
WHERE  2017-08-01 <= t1.val_from AND
       t1.val_from <= 2017-09-01 AND
       2017-08-01 <= t2.val_from AND
       t2.val_from <= 2017-09-01 AND
       t2.val_from > t1.val_from AND
       Interval(t1.val_from, t2.val_from).lengthAtMost(Minutes(7)) AND
       s1.capacity = s2.capacity AND
       s1.capacity > s3.capacity AND
       s1.tx_to>2020-01-01
```

Listing A.30: Query c5b (middle)

```
MATCH  (s3)<-[t2]-(s1)-[t1]->(s2)
WHERE  2017-08-01 <= t1.val_from AND
       t1.val_from <= 2017-09-01 AND
       2017-08-01 <= t2.val_from AND
       t2.val_from <= 2017-09-01 AND
       t2.val_from > t1.val_from AND
       Interval(t1.val_from, t2.val_from).lengthAtMost(Seconds(7)) AND
       s1.capacity = s2.capacity AND
       s1.capacity = s3.capacity AND
       s1.tx_to>2020-01-01
```

Listing A.31: Query c5b (high)

```
MATCH  (s3)<-[t2]-(s1)-[t1]->(s2)
WHERE  2017-08-01 <= t1.val_from AND
       t1.val_from <= 2017-09-01 AND
       2017-08-01 <= t2.val_from AND
       t2.val_from <= 2017-09-01 AND
       t2.val_from > t1.val_from AND
       Interval(t1.val_from, t2.val_from).lengthAtMost(Minutes(60)) AND
       s1.capacity >= s2.capacity AND
       s1.capacity >= s3.capacity AND
       s1.capacity >= 25 AND
       s2.capacity >= 25 AND
       s3.capacity >= 25 AND
       s1.tx_to>2020-01-01
```

Listing A.32: Query c5c (low)

```
MATCH  (s3)<-[t2]-(s1)-[t1]->(s2)
WHERE  2017-08-01 <= t1.val_from AND
       t1.val_from <= 2017-08-08 AND
       2017-08-01 <= t2.val_from AND
       t2.val_from <= 2017-08-08 AND
       t2.val_from > t1.val_from AND
       Interval(t1.val_from, t2.val_from).lengthAtMost(Minutes(10)) AND
       s1.capacity > s2.capacity AND
```

```
        s1.capacity > s3.capacity AND
        s1.capacity >= 50 AND
        s2.capacity >= 37 AND
        s3.capacity >= 37 AND
        s1.tx_to>2020-01-01
```

Listing A.33: Query c5c (middle)

```
MATCH  (s3)<-[t2]-(s1)-[t1]->(s2)
WHERE  2017-08-01 <= t1.val_from AND
       t1.val_from <= 2017-08-02 AND
       2017-08-01 <= t2.val_from AND
       t2.val_from <= 2017-08-02 AND
       t2.val_from > t1.val_from AND
       Interval(t1.val_from, t2.val_from).lengthAtMost(Seconds(25)) AND
       s1.capacity > s2.capacity AND
       s1.capacity > s3.capacity AND
       s1.capacity >= 55 AND
       s2.capacity >= 42 AND
       s3.capacity >= 42 AND
       s1.tx_to>2020-01-01
```

Listing A.34: Query c5c (high)

```
MATCH  (s3)<-[t2]-(s1)-[t1]->(s2)
WHERE  t1.val.lengthAtLeast(Seconds(0)) AND
       t1.val.lengthAtMost(Hours(10)) AND
       t2.val.lengthAtLeast(Seconds(0)) AND
       t2.val.lengthAtMost(Hours(10)) AND
       s1.capacity <= 50000 AND
       s2.capacity <= 50000 AND
       s3.capacity <= 500000 AND
       s1.tx_to>2020-01-01 AND
       2017-08-01 <= t1.val_from AND
       t1.val_from <= 2017-08-08 AND
       2017-08-01 <= t2.val_from AND
       t2.val_from <= 2017-08-08 AND
       s1.capacity >= 50 AND
       s2.capacity <= 25 AND
       s3.capacity <= 25
```

Listing A.35: Query c5a (redundancies)

```
MATCH  (s3)<-[t2]-(s1)-[t1]->(s2)
WHERE  t1.val.lengthAtLeast(Seconds(0)) AND
       t1.val.lengthAtMost(Hours(10)) AND
       t2.val.lengthAtLeast(Seconds(0)) AND
       t2.val.lengthAtMost(Hours(10)) AND
       t2.starttime!=t1.starttime AND
       Interval(t1.val_from, t2.val_from).lengthAtLeast(Seconds(0)) AND
```

```
        s1.id!=s2.id AND
        s1.id!=s3.id AND
        2017-08-01 <= t1.val_from AND
        t1.val_from <= 2017-09-01 AND
        2017-08-01 <= t2.val_from AND
        t2.val_from <= 2017-09-01 AND
        t2.val_from > t1.val_from AND
        Interval(t1.val_from, t2.val_from).lengthAtMost(Minutes(30)) AND
        s1.capacity >= s2.capacity AND
        s1.capacity >= s3.capacity AND
        s1.tx_to>2020-01-01
```

Listing A.36: Query c5b (redundancies)

```
MATCH  (s3)<-[t2]-(s1)-[t1]->(s2)
WHERE  t1.val.lengthAtLeast(Seconds(0)) AND
       t1.val.lengthAtMost(Hours(10)) AND
       t2.val.lengthAtLeast(Seconds(0)) AND
       t2.val.lengthAtMost(Hours(10)) AND
       t2.starttime!=t1.starttime AND
       Interval(t1.val_from, t2.val_from).lengthAtLeast(Seconds(0)) AND
       s1.id!=s2.id AND
       s1.id!=s3.id AND
       s1.capacity <= 50000 AND
       s2.capacity <= 50000 AND
       s3.capacity <= 500000 AND
       s1.tx_to>2020-01-01 AND
       2017-08-01 <= t1.val_from AND
       t1.val_from <= 2017-09-01 AND
       2017-08-01 <= t2.val_from AND
       t2.val_from <= 2017-09-01 AND
       t2.val_from > t1.val_from AND
       Interval(t1.val_from, t2.val_from).lengthAtMost(Minutes(60)) AND
       s1.capacity >= s2.capacity AND
       s1.capacity >= s3.capacity AND
       s1.capacity >= 25 AND
       s2.capacity >= 25 AND
       s3.capacity >= 25 AND
       s1.tx_to>2020-01-01
```

Listing A.37: Query c5b (redundancies)

B.2. Cardinalities and Runtimes

	SF1	SF10	SF100
c0	-	-	1174
c1	632	6668	65462
c2	-	-	1215
c3	1124	130382	12294676
c4	1088	129410	12311535
c5	-	-	12267234
c6	-	-	14254172
c7	15	19164	14633859
c5a (l)	-	-	5117478
c5a (m)	6	504	43136
c5a (h) = c5a	-	-	162
c5b (l)	-	-	4868340
c5b (m)	4	363	37247
c5b (h) = c5b	-	-	229
c5c (l)	-	-	4832533
c5c (m)	5	538	47751
c5c (h) = c5c	-	-	235

Table A.4.: Cardinalities of citibike queries

	SF1	SF10	SF100
c0	-	-	42
c1	14	18	64
c2	-	-	68
c3	15	26	300
c4	15	22	289
c5	-	-	293
c6	-	-	67
c7	15	21	441
c5a (l) = c5a	-	-	228
c5a (redund.)	-	-	234
c5a (m)	15	19	70
c5a (h)	-	-	67
c5b (l) = c5b	-	-	1098
c5b (redund.)	-	-	1403
c5b (m)	15	19	931
c5b (h)	-	-	110
c5c (l) = c5c	-	-	792
c5c (redund.)	-	-	1014
c5c (m)	15	19	76
c5c (h)	-	-	66

Table A.5.: Runtimes of citibike queries (Parallelity 96)

	6	12	24	48	96
c1	410 (1.)	214 (1.92)	135 (3.04)	80 (5.13)	64 (6.41)
c6	518 (1.)	267 (1.94)	169 (3.07)	99 (5.23)	67 (7.73)
c5a (h)	494 (1.)	292 (1.69)	176 (2.81)	98 (5.04)	67 (7.37)
c5b (h)	683 (1.)	381 (1.79)	267 (2.56)	175 (3.90)	106 (6.44)
c5c (h)	450 (1.)	236 (1.91)	139 (3.24)	88 (5.11)	66 (6.82)

Table A.6.: Runtimes and speedup of citibike queries for different degrees of parallelity (SF100)

B. User Guide to Temporal Pattern Matching

A. Pattern Matching in GrALa

To execute a temporal query, the `match` function must be invoked on the data graph. The data graph is an instance of `TemporalGraph.java`. Currently, the API provides several versions of `match`:

- `TemporalGraphCollection query(String query)`

  Execute the GDL query given in `query` on the caller without using statistics[12]. The matching paradigm is isomorphism.

- `TemporalGraphCollection query(String query, String constructionPattern)`

  Execute the GDL query given in `query` on the caller without using statistics. Use the pattern `constructionPattern` to transform the results. The matching paradigm is isomorphism.

- `TemporalGraphCollection query(String query,`
  `TemporalGraphStatistics graphStatistics)`

  Execute the GDL query given in `query` on the caller using the statistics provided by `graphStatistics`. The matching paradigm is isomorphism.

- `TemporalGraphCollection query(String query, String constructionPattern,`
  `TemporalGraphStatistics graphStatistics)`

  Execute the GDL query given in `query` on the caller using the statistics provided by `graphStatistics`. Use the pattern `constructionPattern` to transform the results. The matching paradigm is isomorphism.

- `TemporalGraphCollection query(String query, String constructionPattern,`
  `TemporalGraphStatistics graphStatistics)`

  Execute the GDL query given in `query` on the caller using the statistics provided by `graphStatistics`. Use the pattern `constructionPattern` to transform the results. The matching paradigm is isomorphism.

- `TemporalGraphCollection query(String query, boolean attachData,`
  `MatchStrategy vertexStrategy,MatchStrategy edgeStrategy,`
  `TemporalGraphStatistics graphStatistics)`

  Execute the GDL query given in `query` on the caller using the statistics provided by `graphStatistics`. The matching paradigm is set by `verbStrategy` and `edgeStrategy`. This allows a combination of homo- and isomorphism, e.g. vertices could be matched by homomorphism and edges by isomorphism. The parameter `attachData` determines whether data from the data graph should actually be retrieved (by default, this is done).

[12] technically, statistics that set every estimation to a default value are used

- `TemporalGraphCollection query(String query, String constructionPattern, boolean attachData, MatchStrategy vertexStrategy, MatchStrategy edgeStrategy, TemporalGraphStatistics stats)`

  Execute the GDL query given in `query` on the caller using the statistics provided by `graphStatistics`. Use the pattern `constructionPattern` to transform the results. The matching paradigm is set by `verbStrategy` and `edgeStrategy`. This allows a combination of homo- and isomorphism, e.g. vertices could be matched by homomorphism and edges by isomorphism. The parameter `attachData` determines whether data from the data graph should actually be retrieved (by default, this is done).

As of now, there is only one reasonable implementation of `TemporalGraphStatistics.java` that can be used for `stats`: `BinningTemporalGraphStatistics.java`. To create an instance, first a `BinningTemporalGraphStatisticsFactory` must be created. This is done simply by calling the default constructor `BinningTemporalGraphStatisticsFactory()`. There are four possibilites to create a statistics object with such a factory, as explained in Section 6.2.9:

- `fromGraph(TemporalGraph g)`

  Create binning statistics for the graph. Consider every property. The default maximum sample size for reservoir samples is 5000.

- `fromGraph(TemporalGraph g, Set<String> numericalProperties, Set<String> categoricalProperties)`

  Create binning statistics for the graph, but consider only the specified properties. The default maximum sample size for reservoir samples is 5000.

- `fromGraphWithSampling(TemporalGraph g, int sampleSize)`

  Create binning statistics for the graph. Consider every property. The maximum sample size is set to `sampleSize`.

- `fromGraphWithSampling(TemporalGraph g, int sampleSize, Set<String> numericalProperties, Set<String> categoricalProperties)`

  Create binning statistics for the graph, but consider only the specified properties. The maximum sample size is set to `sampleSize`.

B. Syntax (Overview)

Most of the following tables can also be found in Section 4. They are assembled here again in order to serve as a command reference.

Type	Syntax	Remarks
valid-from (variable)	`variable.val_from`	
valid-to (variable)	`variable.val_to`	
tx-from (variable)	`variable.tx_from`	
tx-to (variable)	`variable.tx_to`	
valid-from (global)	`val_from`	implicitly adds *val-from* $\leq$ *val-to*
valid-to (global)	`val_to`	implicitly adds *val-from* $\leq$ *val-to*
tx-from (global)	`tx_from`	implicitly adds tx-from $\leq$ tx-to
tx-to (global)	`tx_to`	implicitly adds tx-from $\leq$ tx-to
Datetime Literal	`Timestamp(YYYY-MM-DDThh:mm:ss)`	
Date Literal	`Timestamp(YYYY-MM-DD)`	sets time to 00:00:00
Minimum	`MIN(t1,...,tn)`	$n \geq 2$, no nesting of MIN/MAX
Maximum	`MAX(t1,...,tn)`	$n \geq 2$, no nesting of MIN /MAX

Table B.1.: Options to create time stamps in GDL queries

Type	Syntax	Remarks
tx interval (variable)	`var.tx`	
valid interval (variable)	`var.val`	
tx interval (global)	`tx`	implicitly adds *tx-from* $\leq$ *tx-to*
valid interval (global)	`val`	implicitly adds *val-from* $\leq$ *val-to*
custom interval	`Interval(t1, t2)`	*t1, t2* time stamps, creates [*t1, t2*], implicitly adds $t1 \leq t2$
merge interval	`i1.merge(i2)`	implicitly adds constraints enforcing overlap of *i1* and *i2*
join interval	`i1.join(i2)`	implicitly adds constraints enforcing overlap of *i1* and *i2*

Table B.2.: Options to create intervals in GDL queries

Syntax	Semantics
`t1 < / <= / = / != / >= / > t2`	$t1 < / \leq / = / \neq \ / \geq / > \ t2$
`t1.before(t2)`	$t1 < t2$
`t1.after(t2)`	$t1 > t2$
`t1.precedes(i)`	$t1 < i_{from}$
`t1.after(i)`	$t1 > i_{to}$
`i.contains(t1)`	$i_{from} \leq t \leq i_{to}$
`i.fromTo(t1, t2)`	$i_{from} < t2 \ \wedge \ i_{to} > t1$
`i.between(t1, t2)`	$i_{from} \leq t2 \ \wedge \ i_{to} > t1$

Table B.3.: Comparisons involving time stamps in GDL queries

GDL Predicate	Semantics
`i1.overlaps(i2)`	$\max\{a, c\} < \min\{b, d\}$
`i1.contains(i2)`	$a \leq c \wedge b \geq d$
`i1.precedes(i2)`	$b \leq c$
`i1.succeeds(i2)`	$b \leq c$
`i1.equals(i2)`	$a = b \wedge c = d$
`i1.immediatelyPrecedes(i2)`	$b = c$
`i1.immediatelySucceeds(i2)`	$a = d$

Table B.4.: Interval relations in GDL: $i1 = [a, b]$, $i2 = [c, d]$

Duration	Syntax
duration of interval i	(syntax of i as described in Section 4.2)
n milliseconds	`Millis(n)`
n minutes	`Minutes(n)`
n hours	`Hours(n)`
n days	`Days(n)`

Table B.5.: Options to create durations in GDL queries

Syntax	Semantics
`i1.longerThan(i2)`	$\text{length}(i1) > \text{length}(i2)$
`i1.lengthAtLeast(i2)`	$\text{length}(i1) \geq \text{length}(i2)$
`i1.shorterThan(i2)`	$\text{length}(i1) < \text{length}(i2)$
`i1.lengthAtMost(i2)`	$\text{length}(i1) \leq \text{length}(i2)$

Table B.6.: Options to compare durations in GDL queries

Erklärung

Ich versichere, dass ich die vorliegende Arbeit mit dem Thema:

Distributed Graph Pattern Matching on Evolving Graphs

selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemässe Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann.

Das elektronische Exemplar stimmt mit den gedruckten überein.

Höchstädt, den 26.09.2020

_____

Lukas Christ