Scalable Management and Analysis of Temporal Property Graphs

Von der Fakultät für Mathematik und Informatik
der Universität Leipzig
angenommene

Dissertation

zur Erlangung des akademischen Grades

Doctor Rerum Naturalium
(Dr. rer. nat.)

im Fachgebiet

Informatik

vorgelegt
von M. Sc. Informatik Christopher Rost
geboren am 06. September 1990 in Querfurt

Die Annahme der Dissertation wurde empfohlen von:

1. Prof. Dr. Erhard Rahm, Universität Leipzig
2. Prof. Dr. Renzo Angles, Universidad de Talca

Die Verleihung des akademischen Grades erfolgt mit Bestehen der Verteidigung am 15. Mai 2024 mit dem Gesamtprädikat *magna cum laude*.

Für Alina.

Abstract

Graphs, as simple yet powerful data structures, play a pivotal role in modeling and analyzing relationships among real-world entities. In the data representation and analysis landscape, graph data structures have established themselves as a fundamental paradigm for modeling and understanding complex relationships in various domains. The intrinsic domain independence, expressiveness, and the wide variety of analysis options based on graph theory have gained significant attention in both research and industry.

In recent years, companies have increasingly leveraged graph technology to represent, store, query, and analyze graph-shaped data. This has been notably impactful in uncovering hidden patterns and predicting relationships within diverse domains such as social networks, Internet of Things (IoT), biological systems, and medical networks. However, the dynamic nature of most real-world graphs is often neglected in existing approaches, which might lead to inaccurate analytical results or an incomplete understanding of evolving patterns within the graph over time.

Temporal graphs, in contrast, are a particular type of graphs that maintain changing structures and properties over time. They have gained significant attention in various domains, from financial networks over micromobility networks to supply chains and biological networks. A majority of these real-world networks are not static but rather exhibit high dynamics, which are rarely considered in data models, query languages, and analyses, although analytical questions often require an evaluation of the network's evolution.

This doctoral thesis addresses this critical gap by presenting a comprehensive study on analyzing and exploring temporal property graphs. It focuses on scalability and proposes novel methodologies to enhance accuracy and comprehensiveness in analyzing evolving graph patterns over time. It also offers insights into real-time querying, addressing various challenges that emerge when the time dimension is treated as an integral part of the graph.

This thesis introduces the *Temporal Property Graph Model (TPGM)*, a sophisticated data model designed for bitemporal modeling of vertices and edges, as well as logical abstractions of subgraphs and graph collections. The reference implementation of this model, namely Gradoop, is a graph dataflow system explicitly designed for scalable and distributed analysis of static and temporal property graphs. Gradoop empowers analysts to construct comprehensive and flexible temporal graph processing workflows through a declarative analytical language. The system supports various analytical temporal graph operators, such as snapshot retrieval, temporal graph pattern matching, time-dependent grouping, and temporal metrics such as degree evolution.

The thesis provides an in-depth analysis of the data model, system architecture, and implementation details of Gradoop and its operators. Comprehensive performance evaluations have been conducted on large real-world and synthetic temporal graphs, providing valuable insights into the system's capabilities and efficiency.

Furthermore, this thesis demonstrates the flexibility of the temporal graph model and its operators through a practical use case based on a call center network. In this scenario, a

TPGM operator pipeline is developed to answer a complex and time-dependent analytical question. We also showcase the *Temporal Graph Explorer (TGE)*, a web-based user interface designed to explore temporal graphs, leveraging Gradoop as a backend. The TGE empowers users to delve into temporal graph dynamics by enabling the retrieval of snapshots from the graph's past states, computing differences between these snapshots, and providing temporal summaries of graphs. This functionality allows for a comprehensive understanding of graph evolution through diverse visualizations. Real-world temporal graph data from bicycle rentals highlight the system's flexibility and configurability of the selected temporal operators.

The impact of graph changes on its characteristics can also be explored by examining centrality measures over time. Centrality measures, encompassing both node and graph metrics, quantify the characteristics of individual nodes or the entire graph. In the dynamic context of temporal graphs, where the graph changes over time, node and graph metrics also undergo implicit changes. This thesis tackles the challenge of adapting static node and graph metrics to temporal graphs. It proposes temporal extensions for selected degree-dependent metrics and aggregations, emphasizing the importance of including the time dimension in the metrics.

This thesis demonstrates that a metric conventionally representing a scalar value for static graphs results in a time series when applied to temporal graphs. It introduces a baseline algorithm for calculating the degree evolution of vertices within a temporal graph, and its practical implementation in Gradoop is presented. The scalability of this algorithm is evaluated using both real-world and synthetic datasets, providing valuable insights into its performance across diverse scenarios.

Such time series data can also be captured from the application scenario as properties of nodes and edges, such as sensor readings in the IoT domain. In light of this, we showcase significant advancements, including an extended version of the TPGM that supports time series data in temporal graphs. Additionally, we introduce a temporal graph query language based on Oracle's language PGQL to facilitate seamless querying of time-oriented graph structures. Furthermore, we present a novel continuous graph-based event detection approach to support scenarios involving more time-sensitive use cases.

The increasing frequency of graph changes and the need to react quickly to emerging patterns leads to the need for a unified declarative graph query language that can evaluate queries on graph streams. To address the increasing importance of real-time data analysis and management, the thesis introduces the syntax and semantics of *Seraph*, a Cypher-based language that supports native streaming features within property graph query languages. The semantics of Seraph combine stream processing with property graphs and time-varying relations, treating time as a first-class citizen. Real-world industrial use cases demonstrate the usage of Seraph for graph-based continuous queries.

After evaluating lessons learned from the development and research on Gradoop, a dissertation summary and an outlook on future work are given in a final section. This doctoral thesis comprehensively examines scalable analysis and exploration techniques for temporal property graphs, focusing on Gradoop and its system architecture, data model, operators, and performance evaluations. It also explores the evolution of node and graph metrics and the theoretical foundation of a real-time query language, contributing to the advancement of temporal graph analysis in various domains.

Acknowledgments

A special thanks goes to Prof. Dr. Erhard Rahm for supervising this dissertation. He supported me with all underlying publications and gave me the opportunity to work on this topic for nearly 6 years at the database department of the University of Leipzig. I benefited from his many years of research experience, which he always shared with me and the other PhD students, thus significantly improving the quality of our research.

I would also like to thank the co-authors of my publications, especially Andreas Thor, Peter Christen, Angela Bonifati, and Riccardo Tommassini, for helping me write scientifically sound papers. I would also like to thank the Oracle team, especially Dieter Gawlick and Souri, for their support during the collaborative project in 2020. Another thanks go to the Neo4j team for discussing and supporting my last paper about Seraph.

Further, I would like to thank my colleagues in the database department, with whom I had lively and helpful discussions at our regular coffee meetings. My former colleague Kevin Gómez became one of my best and loyal friends during our work on graphs and Gradoop- thank you for the memorable time in our shared office and for teaching me all the basics of graph processing. I would also like to thank Andrea, our database mom, for her help with personal and office matters and encouraging stories about her cats and the conflict between cars and bicycles in Leipzig's traffic. I am also very grateful to my circle of friends, especially my former fellow students and colleagues from Leipzig, who regularly had to listen to my complaints about my lack of progress with the dissertation during our regular movie evenings.

Lastly, I want to thank my family. They were always there for me and supported me psychologically on the way to my dissertation. A big thank you goes to my parents, Anka and Burkhard, and my brother Andy, who supported me through all the ups and downs and always found time to relax on weekends in my former home village. My final thanks go to my dear wife and daughter. Dijana and Alina, I can't even say how much I love and appreciate you two. Thank you for always supporting me in all my plans and sticking with me even when I'm away for one or more weeks on business or private bike trips.

Leipzig, 18. Dezember 2023 Christopher Rost

Dissertation-related Publications

The following list is ordered ascending by publication year.

- **Christopher Rost**, Andreas Thor, Philip Fritzsche, Kevin Gómez, and Erhard Rahm. "Evolution Analysis of Large Graphs with Gradoop". In: *Machine Learning and Knowledge Discovery in Databases - International Workshops of ECML PKDD 2019, Würzburg, Germany, September 16-20, 2019, Proceedings, Part I*, ed. by Peggy Cellier et al. Vol. 1167. Communications in Computer and Information Science. Springer, 2019, pp. 402–408. ISBN: 978-3-030-43822-7. DOI: 10.1007/978-3-030-43823-4_33

- **Christopher Rost**, Andreas Thor, and Erhard Rahm. "Temporal Graph Analysis using Gradoop". In: *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Workshopband*, ed. by Holger Meyer et al. Vol. P-290. LNI. Gesellschaft für Informatik, Bonn, 2019, pp. 109–118. ISBN: 978-3-88579-684-8. DOI: 10.18420/BTW2019-WS-11

- **Christopher Rost**, Andreas Thor, and Erhard Rahm. "Analyzing Temporal Graphs with Gradoop". In: *Datenbank-Spektrum* 19.3 (2019), pp. 199–208. DOI: 10.1007/S13222-019-00325-8

- Kevin Gómez, Matthias Täschner, M. Ali Rostami, **Christopher Rost**, and Erhard Rahm. "Graph Sampling with Distributed In-Memory Dataflow Systems". In: *Datenbanksysteme für Business, Technologie und Web (BTW 2021), 19. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 13.-17. September 2021, Dresden, Germany, Proceedings*, ed. by Kai-Uwe Sattler et al. Vol. P-311. LNI. Gesellschaft für Informatik, Bonn, 2021, pp. 303–312. ISBN: 978-3-88579-705-0. DOI: 10.18420/BTW2021-15

- **Christopher Rost**, Kevin Gómez, Philip Fritzsche, Andreas Thor, and Erhard Rahm. "Exploration and Analysis of Temporal Property Graphs". In: *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021*, ed. by Yannis Velegrakis et al. OpenProceedings.org, 2021, pp. 682–685. ISBN: 978-3-89318-084-4. DOI: 10.5441/002/EDBT.2021.83

- **Christopher Rost**, Philip Fritzsche, Lucas Schons, Maximilian Zimmer, Dieter Gawlick, and Erhard Rahm. "Bitemporal Property Graphs to Organize Evolving Systems". In: *CoRR* abs/2111.13499 (2021). arXiv: 2111.13499

- **Christopher Rost**, Kevin Gómez, Matthias Täschner, Philip Fritzsche, Lucas Schons, Lukas Christ, Timo Adameit, Martin Junghanns, and Erhard Rahm. "Distributed temporal graph analytics with Gradoop". In: *VLDB J.* 31.2 (2022), pp. 375–401. DOI: 10.1007/S00778-021-00667-4

- **Christopher Rost**, Kevin Gómez, Peter Christen, and Erhard Rahm. “Evolution of Degree Metrics in Large Temporal Graphs”. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2023), 20. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 06.-10, März 2023, Dresden, Germany, Proceedings*, ed. by Birgitta König-Ries et al. Vol. P-331. LNI. Gesellschaft für Informatik e.V., 2023, pp. 485–507. ISBN: 978-3-88579-725-8. DOI: 10.18420/BTW2023-23

- **Christopher Rost**, Riccardo Tommasini, Angela Bonifati, Emanuele Della Valle, Erhard Rahm, Keith W. Hare, Stefan Plantikow, Petra Selmer, and Hannes Voigt. “Seraph: Continuous Queries on Property Graph Streams”. In: *Proceedings of the 27td International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy, March 25 - March 28, 2024*, ed. by Letizia Tanca et al. OpenProceedings.org, 2024. DOI: 10.48786/edbt.2024.21

Contents

List of Figures

List of Tables

Part I

Foundations

1 Introduction

"Large-scale temporal graphs are everywhere in our daily life." - Dr. Bo Zong (2015)

This first chapter motivates the importance of the temporal dimension for graphs in Section 1.1, especially the benefits of native support of dynamics for graph data models, query languages, metrics, and analytics. After a subsequent summary of the challenges in this field of research, the scientific contributions of this dissertation are specified in Section 1.2 whereas Section 1.3 gives an overview of the thesis structure.

1.1 Motivation

Graphs are a powerful data structure that can be used to represent complex relationships between objects in a variety of domains [219], ranging from social networks [90], financial transaction networks [162, 163], genome interactions [24], chemical compounds [188] to electrical infrastructure [8]. The use of graphs as a data representation and analysis tool has gained fundamental attention in recent years, driven by the growing availability of large-scale network datasets [187] and the need to extract meaningful insights from them. Noteworthy contributions in the domain have been made by both academic institutions and major tech companies, such as Google, Amazon, Facebook, and Microsoft, who have introduced diverse systems dedicated to the management and processing of graphs [189]. According to Gartner, there is a projected exponential growth in the adoption of graph technology, with an anticipated 80 percent incorporation into data and analytics developments by 2025, a substantial increase from the 10 percent recorded in 2021 [21].

Unlike other data models, graphs are specifically designed to represent relationships (called “edges”) between entities (called “vertices”) and, thus, recognize patterns that are difficult to discern in other representations, such as tables. Graphs provide a flexible and intuitive framework for modeling complex relationships and dependencies in data, making them a valuable asset for researchers and practitioners. Many algorithms can be executed on graphs, including graph traversal algorithms, shortest path algorithms, and graph clustering algorithms, among others, all of which offer powerful ways to analyze and understand complex relationships in data.

However, the vast majority of graph databases, graph query languages, and processing systems only consider traditional static graph data and neglect one crucial aspect:

time.

In contrast to *static* graphs, where time and changes only play a subordinate role, so-called *dynamic graphs* treat real-world networks' dynamic nature as a data structure's central feature. A special kind of dynamic graphs are *temporal graphs*, which maintains the history of a graph's changes over an observed period. Most real-world graphs change over time and are thus implicitly temporal graphs if the information about when the changes happen is available from the real-world observation. Let's discuss some applications.

Social Networks, like X (former Twitter), Facebook, or Instagram, is a prime example of temporal graphs. New users continually join the network, forming connections through friendships and follows, and communities evolve over time. Tracking the temporal aspects of user activities, such as new posts and follows/unfollows, is essential for uncovering patterns in user engagement and community dynamics. Moreover, the influence of users on the network changes over time, making it imperative to consider the evolution of the graph to capture the shifting landscape of social interactions accurately. This temporal perspective is fundamental for comprehensively analyzing social network structures and behaviors.

Supply Chains model the way of products that undergo a series of processes from production to selling, and analyzing the chain's evolution over time is essential for efficient management. Temporal graph analysis enables the study of the flow of products, the identification of bottlenecks, and the optimization of processes. By recognizing how the supply chain transforms over time, organizations can make informed decisions to enhance productivity and adapt to changing market demands, ensuring a more resilient and responsive supply chain. The ability to visit the network at a previous stage also provides a retrospective view of decisions made in the past.

Financial Networks build another use case where temporal graph analysis plays a critical role, especially in detecting and preventing financial fraud. Money transfers, the creation of new accounts and devices, and transaction patterns are all temporal. If the temporal development of activities is considered, detecting anomalies and fraud patterns can be improved compared to observing purely static patterns. The chronological order of events, such as the sequence of transactions, provides an intuitive way of representing fraudulent behavior.

Micromobility Networks model bike, car, and scooter rentals in urban environments. Crucial for urban sustainability, understanding the network's temporal evolution is essential. The availability of rental vehicles, changes in the network structure, and the introduction of new vehicles and stations all occur over time. Temporal graph analysis helps predict demand, conduct historical analyses, and optimize the distribution of resources to make informed decisions and improve service efficiency, enhance user experience, and contribute to a more sustainable urban environment. We provide a more detailed example of this domain later in this section.

In the last years, research focus on the evolution of graphs by treating time as an essential part of the modeling, querying and analysis [130, 138, 139, 145, 197, 215]. Using a *temporal* graph model instead of a static one, the evolving interconnectivity of entities is

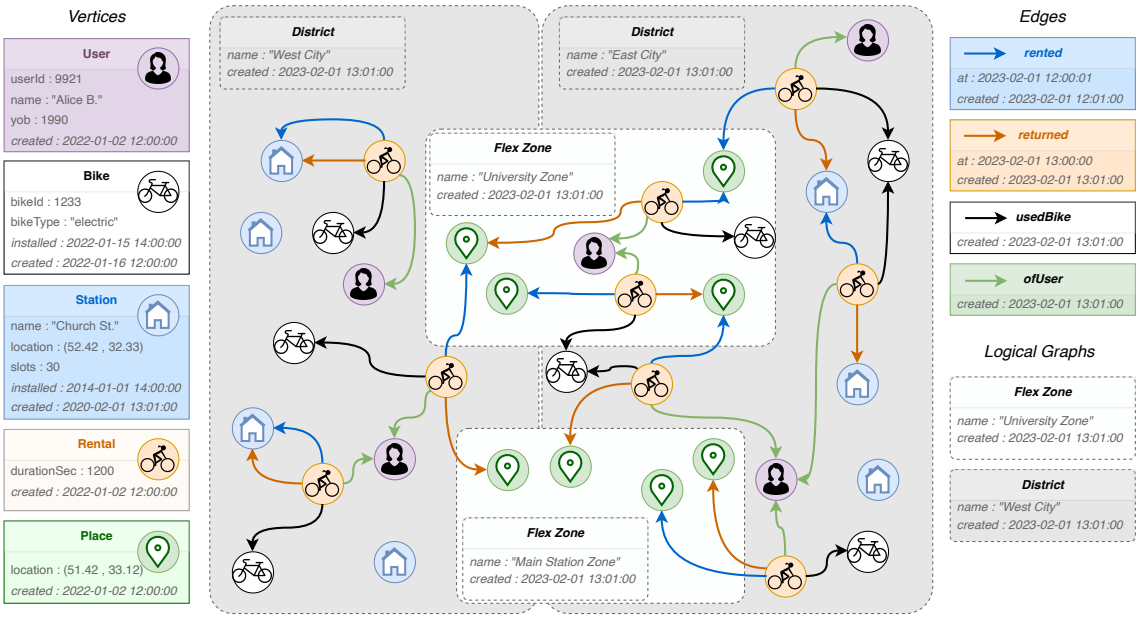


Figure 1.1: Example of a bike-sharing graph. Additional zones and districts add further semantics to the graph elements.

captured and thus unlocks a surprisingly large domain of analysis opportunities focused on the time dimension [197]. Existing graph algorithms and metrics can be extended to take the graphs temporal evolution into account, e.g., temporal shortest paths [221] or foremost paths [101] instead of shortest paths. But also completely new analyses can be performed, e.g., the calculation of a difference graph [175], which shows structural differences between two temporal states of the graph. The research landscape around temporal graphs and their storage and processing presents a wide range of challenges, which we will explore below using a real-world example from the micromobility domain.

Bike-sharing systems, ubiquitous in urban environments, exhibit a rich temporal structure as users engage in time-varying bike rentals and return patterns. Entities such as bikes of different types, users with different subscriptions, rental stations in different locations, and rental zones are connected by highly dynamic relationships. A temporal graph is the ideal representation of the development of such a highly dynamic network. The graph models the relationships between the entities and, at the same time, the time when and for how long they were valid or when information about their validity was inserted into the graph.

Figure 1.1 shows an example of a bike-sharing graph. As in all transformations to a graph data structure, different graph schemas can be chosen to model the same use case. For this example, we chose five types of vertices (see the examples on the left side of Figure 1.1), four types of edges (see top right), and two types of logical abstractions of subgraphs (see bottom right), which we later define as so-called *logical graphs* (see Section 3.3).

The vertices represent the real-world entities in our graph: users, bicycles, rental stations, the rental itself and locations within a flex zone where a bike can be parked. Each vertex is described in more detail by further properties, such as identifiers or geo-coordinates for locations and rental stations. The edges represent the relationships between the entities, e.g., *rented* assigns a rental vertex to the corresponding start location

whereas *returned* assigns it to the end location. Flex zones (places where bicycles can be returned without a rental station) and districts are logical graphs that help assign vertices and edges to logically possibly overlapping subgraphs. For example, in the flex zone "University Zone" are four *Place* vertices that are also partially part of districts. The two places on the left side of the zone are also part of the district "West City", whereas the two on the right are also part of the district "East City".

The graph is also enriched with various temporal data about its development. All graph elements initially have a property *createdAt*, which contains the time when the information about the element was inserted into the graph. We later refer to this time dimension as system time or transaction time.

Furthermore, temporal information about their validity from real-world entities is available: When was a rental station installed? How long has a bike been in use? When did a user register, and is he or she deactivated again? When did the rental take place, and when was the bike returned? We later refer to this time information as the valid time or application time.

This example gives rise to the first challenge for the analysis of temporal graphs: a rich temporal graph data model.

Challenge C1. ***Temporal graph data model.*** *The development of a rich temporal graph data model that can represent both structural and content changes is an important research challenge. It should support nodes, edges and logical (sub)graphs to which nodes and edges can be assigned. All three graph element types should support type labels and optional properties to add further semantics. The temporal evolution of the elements should be a first-class citizen of the data model to enable a more comprehensive understanding of dynamic relationships in various domains.*

The principle of extending a static data model with time dimensions to a temporal one already existed in the relational world. So-called *temporal tables* [120], standardized with SQL:2011, contain extra columns for storing historical information about the development of the data they contain. What stands out in this modeling is the addition of not only a single time dimension (unitemporal) but the possibility to choose a bitemporal modeling [105], i.e., *two* orthogonal time dimensions. A distinction is made between the valid time (application time) and the transaction time (system time) [105]. The former describes the time when a fact occurred in the real world and how long this information is valid. The transaction time, in turn, describes when this information was visible to the system storing the data, i.e., when it was inserted or (logically) deleted.

This approach of adding two time intervals to the data structure can also be applied to graphs. In some cases, companies must comply with specific legal requirements that require accurate recording of data changes. For our bike-sharing graph, we also have this bitemporal data available. For example, the edges of type *rented* and *returned* have two times: when the rental/return happened in the real world and when it was inserted into the graph. This could be important for more accurate billing or changing billing or reconstructing predicted demands. However, bitemporal modeling in the graph domain is also applicable in more critical domains such as finance, supply chains, or medical research whenever the history of the real world and the database are both critical. For example, a bank denied a person's creditworthiness at a certain point in time because the data on a sufficiently funded account was only transmitted with a delay. Or in

medical research, bitemporal modeling can help track the progression of diseases along a path in the graph and medical interventions over time. It enables the analysis of patient trajectories, treatment effects, and identifying trends or path patterns that may be relevant for improving healthcare. Both time dimensions can always be included, e.g., when exactly was a medication administered and under which state of knowledge? This leads to the next challenge: integrating a bitemporal time model.

Challenge C2. ***Bitemporal modeling.*** *A bitemporal modeling provides two time semantics using two orthogonal time domains: the valid time (also called application time) to distinguish when and how long a fact occurred in the observed real world and transaction time (also called system time) to maintain when and how long the information about this occurrence is available in the graph storage. A challenge is the integration of these two orthogonal time dimensions into the graph data model and its use within queries, analyses, and algorithms.*

In the last decade, many practical, declarative query languages have been developed for querying instances of graph data models. The two most fundamental graph querying functionalities are graph patterns and navigational expressions [15]. Graph patterns match a graph-structured query against the data, whereas navigational expressions match patterns recursively against the graph to navigate paths of arbitrary length. For a temporal bike-sharing graph like the one in Figure 1.1, a query should satisfy two criteria: firstly, it must be possible to use all time information for projection, selection, and return, and secondly, it should be possible to formulate chronological orders in the pattern. A query could, for example, ask for traces of a user's path through rented bikes and rental stations over time or analyze the temporal utilization of flex zones by tracking the count of bikes parked and their average duration within each flex zone over time. Another query could explore how the connectivity between rental stations evolves, focusing on the stations that experience the most frequent changes in incoming or outgoing bike traffic. Executing a query on different graph states is also a challenge to answer questions like "What is the result of my graph query, and what would the result be for the same query if I had asked it last week, last month, or last year?"

The need to extend graph query languages to include temporal dimensions leads to the next challenge for temporal graph research: a temporal query language.

Challenge C3. ***Temporal graph querying.*** *Developing efficient temporal graph query languages and evaluation mechanisms is an important research challenge. In addition to the well-established graph patterns and navigational expressions, queries should support the projection, selection, and return of all available temporal attributes. Further, it should be easy to formulate historical queries, i.e., executing a query at a past graph state and temporal patterns and paths.*

There is much more to analyzing graphs than just querying for patterns or paths. The graph data structure allows a variety of further analyses, such as detecting communities or frequent patterns, centrality or connectivity analysis, or path analysis like Single Source Shortest Paths (SSSP). Such essential graph analyses are indispensable tools for graph analysis systems.

However, if applied on a temporal graph, the challenge is to consider the time dimension within the respective algorithm. For example, suppose a community detection algorithm,

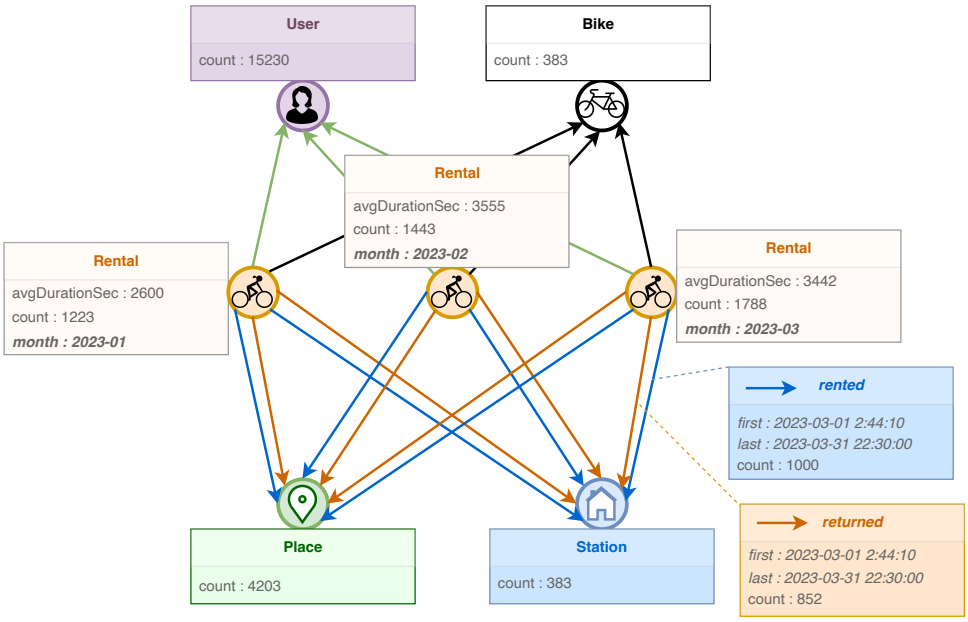


Figure 1.2: Summarized bike-sharing graph where the rentals are monthly grouped.

like the Louvain Algorithm [137], performs a static assignment of vertices to a cluster. In that case, this assignment will most likely change as edges are added and removed, which leads to the research area of dynamic community discovery [171].

Extending a static graph analysis to a temporal one is essential, but also the analysis and exploration of the development of the graph itself. For example, an analyst wants to know how a graph changed between a past an the current state, how frequent changes of a specific vertex/edge type occurred, or how the average duration of relationships between two vertex types changed over time. This brings us to the next challenge in analyzing temporal graphs:

Challenge C4. ***Temporal graph analysis.*** *The flexible analysis of temporal graphs includes all the analysis capabilities of a static graph under consideration of the graph evolution, as well as the analysis of the evolution and dynamics of the graph itself. Developing flexible, extendable, and domain-independent temporal graph analysis techniques is a significant research challenge.*

Temporal graphs also tend to be very large, especially when looking at a high-frequent graph evolution over a long period, such as several years. Here, it may be beneficial to group the graph to get an overview of the evolution of similarities and differences in the graph, reduce the number of nodes and edges to simplify it, gain deeper insights into the structural dynamics, or reduce the granularity of the graph's evolution. Such grouping (also called summarization), essential in the relational world and already studied for static graphs [111], is part of current research for temporal graphs. Again, the challenge is considering the time dimension in the summarization method. Grouping is performed not only for elements that are similar in structure or content but also for elements with similar temporal properties, such as being valid on the same day of the week, having a similar duration, or being close in time.

For example, if the temporal graph in Figure 1.1 is summarized by grouping nodes and edges with the same label and that are also valid in the same year and month, the result

would exemplary look like Figure 1.2. One can see that all rental vertices of a specific month are grouped as one vertex (for simplicity, we just showed the result for 3 months of a year), which is shown in the middle of the figure. Each is augmented with aggregated values, e.g., the average duration of all rentals and the count. The remaining vertex types are on the top and bottom of the figure, again with a count property. Edges are grouped according to their label and annotated with an edge's first and last occurrence and the count. An analyst can thus get deeper insights into the evolution of the graph at different temporal granularities. A system for analyzing temporal graphs should support such grouping, which leads to the next challenge.

Challenge C5. ***Temporal graph summarization.*** *Temporal graphs can be large and complex due to the evolution of the graph elements as part of the graph model. Developing techniques to summarize a temporal graph by its structure, content, and evolution is a current research challenge to compute smaller, condensed, and possibly aggregated graph versions that offer deeper insights into the graph's evolution.*

Typical graph analyses also include calculating and evaluating centrality measures. Centrality is a fundamental concept in identifying important vertices in a graph. A centrality measure (or metric) quantifies the importance of a vertex, depending on how "importance" is defined. There are a variety of such centrality measures, from degree centrality to betweenness centrality to the popular page rank. Such a metric results in a static score for static graphs, but for temporal graphs, the graph's structure changes over time, which also may change the metrics. Thus, a vertex metric no longer represents a scalar value but is a time series for temporal graphs [174], revealing hidden information like periodical patterns or peaks at certain times. This problem leads to the next challenge for temporal graph analysis:

Challenge C6. ***Temporal graph metrics.*** *The temporal evolution of a vertex metric reveals important patterns that would otherwise remain hidden in the static case. An important challenge is defining and developing new temporal metrics and extending static metrics with the available time dimension.*

Analytical questions can be very complex. Thus, a single query or a specific graph analysis operation can be insufficient to achieve the desired result. In addition to offering a variety of analysis operators and algorithms, a system that enables the analysis of temporal graphs should support the construction of analytical pipelines. These pipelines allow concatenating multiple operations, e.g., a subgraph extraction followed by a pattern matching query with a final community detection algorithm. Providing operator composition would address a wide range of use cases and answer diverse questions that a single query or analysis can not answer.

One aim of analyzing the bike-sharing graph in Figure 1.1 could be to predict the demand for bicycles at stations and flex zones for the coming year. Such a question can hardly be answered by a single query or by a single algorithm but requires the combination of various successive steps. Figure 1.3 shows an example of such a pipeline.

Let's briefly go through the figure step by step. Starting from the temporal bike-sharing graph, we first apply a graph grouping (1) to group all place vertices in flex zones to artificial vertex stations. Then, we extracted two temporal graph snapshots (2) that represent the evolution of 2021 and 2022 to predict later the demand for 2023.

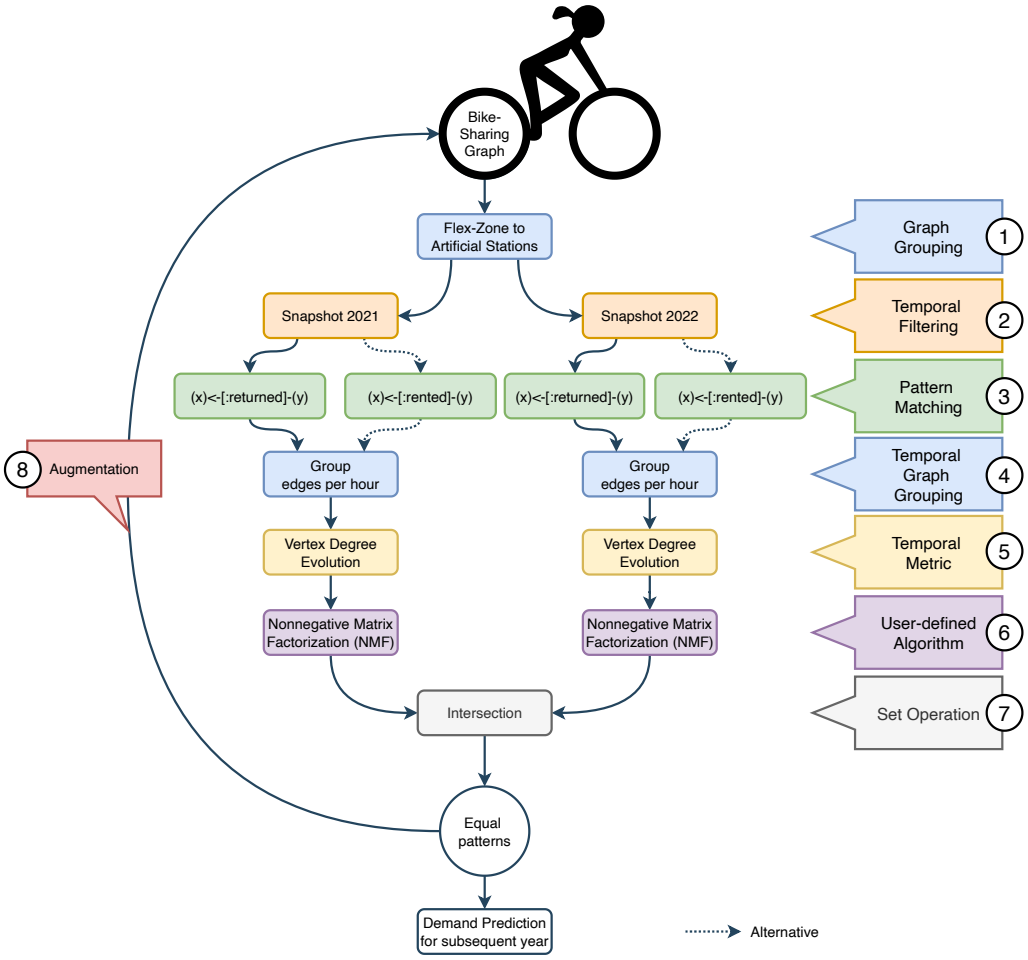


Figure 1.3: Example of an analytical pipeline to predict the demand of bikes at stations and flex zones for the following year.

In a subsequent step, we use a pattern-matching query (3) to match all *return* edges to predict later how many bikes will be returned. In a later step, the whole pipeline can be executed again by an alternative pattern that queries all *rental* edges to approximate the rentals. After we temporally group (4) the resulting edges per hour of the day, we calculate the evolution of each vertex degree metric (5) to get a time series of the bike returns for each station. One way to extract temporal patterns from these time series is using Nonnegative Matrix Factorisation (NMF) [164]. The rental station's quantitative affiliation of the time series to a temporal pattern retrieved through NMF (6) (e.g., typical business hours or weekend-focused) characterizes the rental station.

Since we have applied all intermediate steps on both the 2021 and 2022 rentals, an intersection (7) can now be applied to both results to determine which stations show a consistent pattern for both years. This can then be taken as an indication of the stability of behavior and can be used to predict demand for the following year. The analysis result can also be used to augment the initial graph (8) with this information.

Such a possibility of forming analytical workflows thus represents an essential challenge for temporal graph analysis systems.

Challenge C7. ***Analytical pipelines.*** *In addition to offering a variety of analysis operators*

and algorithms, a challenge is to provide the ability to compose analytical pipelines by flexibly connecting single analysis steps.

One can now imagine that the (bi)temporal extension of a graph, which can already be very large, makes it even more extensive compared to a static graph just representing its state at the most current time. Depending on the performance of a single system (CPU, main memory, disk space), the available resources may no longer be sufficient for a particular graph size or complexity, which means a distributed solution is needed to process and store the entire graph. Sahu et al. show in [187] that distributed graph analysis systems are widely used in companies to respond to growing sizes of graph datasets. Whether the overhead of distributed processing outperforms the increased performance for a specific graph size, the COST-metric [135] can be determined using appropriate benchmarks.

Challenge C8. ***Scalable storage and processing.*** *Temporal graphs can become very large, especially due to the additional time dimension(s) involved. A research challenge is to provide scalable storage for the temporal graph and to guarantee the scalability of queries, analysis, and algorithms to handle this large graph data using distributed and parallel techniques.*

The effective visualization of temporal graphs and analysis results presents a significant challenge due to its dynamic nature, where edges and vertices constantly change over time. A suitable visualization assists in conveying temporal information, such as the emergence and dissolution of connections, evolving network structures, and temporal patterns. It helps identify significant time points, anomalies, and recurring patterns, allowing researchers and analysts to make advanced decisions that might not be possible on a static graph view. This allows us to formulate the next challenge.

Challenge C9. ***Temporal graph visualization.*** *Visualizing temporal graphs and their analysis results can be more complex and difficult compared to static graphs due to the additional time dimension(s). Developing suitable visualization techniques for temporal graphs is challenging and an important research area.*

Temporal graph analytics focuses on the dynamic nature of evolving systems represented by graphs. While the historical data in a temporal graph offers valuable insights about what already happened, relying solely on the past may overlook emergent patterns and anomalies that manifest in real-time. Quick decision-making is essential in time-critical environments, such as IT security or financial transactions, necessitating the ability to analyze streams of graph updates with low latency. Real-time querying and analysis also enable the development of adaptive systems that respond dynamically to changes. The prevalence of streaming data from sources like sensor networks reinforces the challenge of analyzing temporal graphs as continuous streams to capture and process the most recent information and provide query languages suitable for continuous evaluation.

A plausible implementation in the described bike-sharing scenario entails introducing a real-time bonus program designed to redistribute bicycles dynamically. More precisely, when a user rents a bike, the system assesses the current usage patterns of stations and flex zones. It suggests to the user that they return the bicycle to a station or zone experiencing

high demand but with a low availability of bicycles. If the user accepts this suggestion, they are rewarded with a discount for their ongoing trip. It is imperative to express such a query in a suitable language and consistently evaluate the query concurrently with the updates to the graph that arrive as a stream.

Challenge C10. ***Graph stream processing.*** *Graph data can arrive as a continuous unbounded stream of data, contrasting to the concept of temporal graphs, which maintain the past evolution of graph elements. Developing suitable stream models, efficient stream algorithms, and languages for processing and querying graph streams parallel to new incoming data is an emerging research challenge. Another research challenge in this field is the investigation of graph query operators for path-oriented semantics on graph streams.*

To put everything together, current research on large-scale temporal graph processing and analysis includes various research challenges. We selected ten important challenges that can be finally summarized as follows:

C1 Temporal graph data model. The challenge is to develop a data model that can fully represent the evolution of a graph and treats *time* as a first-class citizen.

C2 Bitemporal modeling. The integration of two orthogonal time dimensions into the graph data model and its usage within queries, analyses and algorithms is a current research challenge.

C3 Temporal graph querying. The development of efficient temporal graph query languages and evaluation mechanisms is an important research challenge.

C4 Temporal graph analysis. Developing flexible, extendable, and domain independent temporal graph analysis techniques considering the graph evolution is an important research problem.

C5 Temporal graph summarization. Developing techniques to summarize a temporal graph by its structure, content, and evolution is a current research challenge to compute smaller, condensed, and possibly aggregated versions of the graph.

C6 Temporal graph metrics. The evolution of a vertex or graph metric reveals important patterns that may otherwise remain hidden in the static case. A challenge is the definition and development of new temporal metrics and the extension of static metrics with the time dimension.

C7 Analytical pipelines. In addition to offering a variety of analysis operators and algorithms, a challenge is to provide the ability to compose analytical pipelines by flexibly connecting single analysis steps.

C8 Scalable storage and processing. A research challenge is to provide scalable storage for the temporal graph and to guarantee the scalability of queries, analysis, and algorithms to handle this large graph data using distributed and parallel techniques.

C9 Temporal graph visualization. Visualizing temporal graphs and their analysis results is a current research challenge due to the additional time dimension(s) involved.

C10 Graph stream processing. Graph data can arrive as a continuous unbounded stream, which needs efficient stream models, algorithms for processing, and languages for querying graph streams in parallel to new incoming data.

This thesis deals with the elaborated ten challenges and shows how developed approaches fulfill them entirely or partially. Besides, many more challenges are ahead,

which are mentioned in the following paragraph for completeness. However, they are not considered further since they are far-reaching and thus would exceed the scope of the work.

Further Challenges beyond this thesis. Another worth mentioning challenge due to the topicality of the subject is ***graph machine learning***, or ***Graph-ML*** for short. Neural networks that operate on graph data (called graph neural networks, or GNNs) were researched for over a decade [191]. The goal of GNNs are mainly *link predicton* [124], *node classification* [222], *graph classification* [124] and *graph reconstruction* [48].

Link prediction aims to predict missing or future connections between vertices (e.g., a bike rental), whereas *node classification* aims to predict the classes of unlabeled nodes as node properties based on other node properties (e.g., to classify bike rental users according to their rental behavior). In *graph classification*, the goal is to learn a classifier that can accurately predict the class or category of a graph according to structural properties (e.g., to classify places to demands to suggest a place for a new rental station), and *graph reconstruction* aims to reconstruct a graph from a set of its subgraphs (e.g., to reproduce lost rental information).

In recent years, GNN-based models for temporal graphs have emerged as a promising research area to extend the capabilities of GNNs [128]. Such Temporal Graph Neural Networks (TGNNs) focus on the time dimension in prediction and classification tasks, e.g., to predict *when* a link will happen or *when* and *how long* a node is assigned to a class. The temporal dynamics of graphs require models that are able to incorporate the time dimension of the graph by creating suitable embeddings, i.e., a mapping of the graph data into a lower-dimensional space.

Another challenge is the development of ***temporal graph schemas*** and ***temporal key constraints***. Graphs are known for being schema-free. However, a schema definition for graphs has advantages related to various concepts, like query optimization, data validation, interoperability, maintenance, and documentation. A schema definition is crucial for structuring data and a desirable feature of graph users, which is shown by a recent survey [187]. Towards this need, a recent work introduced a formalism for specifying (static) property graph schemas, called *PG-Schemas* [18], which allows a user to impose structure on nodes, edges, and properties of the underlying graph instances.

Similar requirements hold for key constraints, whose main goal is enforcing data integrity and allowing the referencing and identifying of objects [17]. Key constraints should apply to nodes, edges, and properties and provide combinations of basic restrictions that require the key to be exclusive, mandatory, and singleton. A first framework for defining keys for (static) property graphs is the recently published work with the title *PG-Keys* [17]. However, a formalism for defining schemas and key constraints on temporal property graphs need extensions to this existing work and is, to the best of our knowledge, still missing and, thus an open challenge.

Another challenge is ***temporal graph clustering***, whose main objective is to identify patterns or trends in the temporal behavior of entities within the graph. Graph clustering can be used for many applications, such as community discovery [171], anomaly detection [132] or social group analysis [62]. The main challenge, according to temporal graph clustering, is incorporating the graph evolution in the algorithms with the goal, of revealing groups of nodes and edges that exhibit similar temporal dynamics and understanding

how these groups interact and evolve.

In addition to the challenges mentioned above, there are many others, such as ***efficient storage, indexing and retrieval*** [133] to enable fast ingestions and access, ***temporal pattern mining*** [31] to identify recurring patterns that frequently appear in the graph intending to identify temporal dependencies, ***change point detection*** [96] to identify points in time with significant structural graph changes, or ***uncertain temporal graph mining*** [146] to develop methods to uncover meaningful patterns or knowledge from graph data that exhibit both temporal dynamics and uncertainty.

1.2 Scientific Contributions and Publications

The contributions of this dissertation refer to the challenges mentioned above. Most of the contributions are published in scientific journals, conference proceedings, or workshops and are thus peer-reviewed, which is accordingly marked by references.

Temporal Graph Model, Analysis Operators, and Query Language: Typically, graphs are viewed as a static construct representing a concrete or aggregated state of interactions between entities of a real-world scenario. However, a temporal graph model is needed if the evolution of the graph needs to be maintained as a first-class citizen. Hence, queries and analyses can use this additional temporal information as an essential component to reveal information hidden in a static view. We therefore extended the existing property graph model EPGM and its analysis operators with a bitemporal model to the Temporal Property Graph Model (TPGM). Besides a rich temporal data model, it offers various temporal analysis operators (e.g., temporal pattern matching by a developed declarative language *TemporalGDL* and time-dependent graph grouping) and an analytical language to compose workflows. The model was integrated completely into the existing distributed Gradoop system. All model, operator, and language concepts, implementations, evaluations as well as lessons learned have been published in three papers, including *Temporal Graph Analysis using Gradoop* [179] published 2019 at the BigDS workshop which was co-located with the BTW 2019, its extended version *Analyzing Temporal Graphs with Gradoop* [178] published in the Datenbank Spektrum (vol. 19), and *Distributed temporal graph analytics with Gradoop* [176] published 2021 in the VLDB Journal. This contribution is far-reaching and addresses the challenges C1, C2, C3, C4, C5, C7 and C8.

Analytical Pipelines and Temporal Graph Exploration: With the TPGM, we offer a flexible way to represent and analyze temporal graphs. Two works have been published to demonstrate the usability by employing real examples. On the one hand, a paper called *Evolution Analysis of Large Graphs with Gradoop* [177] was published in the Large Evolving Graphs (LEG) workshop of the ECML PKDD 2019, which shows, based on a call center network, how the operators of the TPGM can be combined to realize a temporal graph analysis workflow. Furthermore, a web application called *Temporal Graph Explorer (TGE)* was developed and published as a demo paper at the EDBT 2021 under the title *Exploration and Analysis of Temporal Property Graphs* [175]. The TGE allows users to retrieve snapshots of past graph states, compute differences between

graph snapshots, and temporally summarize graphs, thus gaining deeper insights into graph evolution. The respective operator can be flexibly configured, visualizing the result in a cartographic representation. The contributions behind both publications addressing the challenges C4, C7, C5, and C9.

Evolution of temporal graph metrics: In a temporal graph, the graph structure changes over time. New vertices and edges are added, or existing edges are removed or lose their validity. As the graph changes, vertex and graph metrics change over time. A vertex metric, such as the simple and frequently used vertex degree, is a scalar value in a static graph but results in a time series for temporal graphs. In the work *Evolution of Degree Metrics in Large Temporal Graphs* [174], published at the BTW 2023, a set of temporal extensions of four degree-dependent metrics is proposed, as well as aggregations like minimum, maximum, and average degree of (i) a vertex over a time interval and (ii) a graph at a time point. Since using the static degree can lead to wrong assumptions about the relevance of a vertex in a temporal graph, the need to include time as a dimension in the metric is highlighted. Further, a baseline algorithm and its implementation in Gradoop is outlined. This contribution addresses the challenges C4, C6 and C8.

Temporal Graph Query Language and Continuous Graph Notifications: A one-year research cooperation between Oracle Corp. and the University of Leipzig in 2020 aimed to investigate the organization of relationships within multi-dimensional time-series data, particularly sensor data from the IoT area. The project proposed using temporal property graphs, with some extensions, as a suitable approach for this organizational task, combining the strengths of both graph and time-series data models.

The outcome of the research cooperation was published on arXiv as a summarized project report, namely *Bitemporal Property Graphs to Organize Evolving Systems* [173] and includes three major achievements: 1) *TPGM*$^+$: the extension of the bitemporal property graph model TPGM with support for property updates; 2) *T-PGQL* the extension of Oracle's PGQL graph query language by bitemporal operations; and 3) the conception of *CGN* (Continuous Graph Notifications), which is a method for identifying and reacting to significant events in the temporal property graph in real-time. The work presents new graph modeling techniques, a query language, and real-time event detection capabilities that can be used in various applications, particularly in the IoT domain. The contributions of this work addressing the challenges C1, C2, C3 and C10.

Continuous graph query language: Real-time data analysis and management are becoming increasingly important for today's businesses. According to the graph data structure, static query languages like the widely used Cypher form Neo4j lack the features to handle streaming graph data and their continuous query evaluation. In a collaborative work between Neo4j Inc., the Université Lyon 1, and the University of Leipzig, we developed *Seraph*, a Cypher-based continuous graph query language supporting native streaming features.

The respective publication, namely *Seraph: Continuous Queries on Property Graph Streams* [180], accepted at the research track of EDBT'2024, formally defines the language semantics by combining stream processing with property graphs and time-varying relations. Besides formalizing the Seraph syntax, we discuss the potential impact both from

a user and a system perspective and showcase the usage of the language for emerging graph-based continuous queries based on real-world industrial use cases. With this contribution we address the challenges C1, C3 and C10.

The collaboration on the project was also the basis for a joint research project called *HyGraph*, funded by the German DFG and the French ANR for 3 years starting in 2023.

1.3 Structure

This dissertation contains eight further chapters with the following summarized content.

Chapter 2 gives the necessary preliminary knowledge to understand the individual chapters and related work that relates to the entirety of the dissertation. Specific related work to individual chapters is evaluated in the chapters themselves.

Chapter 3 introduces the Temporal Property Graph Model (TPGM), including a bitemporal graph data model, a set of combinable analysis operators, and an analytical language to build analytical workflows. Special attention is paid to *TemporalGDL*, a declarative graph query language for bitemporal property graphs, and a grouping operator, which summarizes temporal graphs based on their temporal characteristics. As a reference implementation of the TPGM, we introduce a temporally extended version of Gradoop, an open-source framework for distributed analysis of large property graphs.

Chapter 4 gives two applications of Gradoop and it's temporal graph analysis features. First, a call center network use case is solved with the TPGM and its combinable operators. Second, a web-based user interface called *Temporal Graph Explorer* is demonstrated, showing the usage of three temporal TPGM operators by real-world temporal graphs. It further offers a visualization of temporal graphs and the analytical results.

Chapter 5 is about the implicit evolution of graph metrics of temporal graphs, specifically, degree-based metrics. Four selected metrics were extended by the temporal dimension and formally described. A prototypical implementation of a baseline algorithm in Gradoop shows that it scales well for large temporal graphs.

Chapter 6 summarizes project results of a one-year cooperation with a leading database company. It defines an extended version of the TPGM data model, namely TPGM$^+$, a temporal extension of the property graph query language PGQL, namely T-PGQL, and a continuous graph notification mechanism called CGN.

Chapter 7 proposes Seraph, a Cypher-based language supporting native streaming features. The data model, language semantics, and syntax are formally defined by combining stream processing with property graphs and time-varying relations. We show the usage of the language for emerging graph-based continuous queries based on three use cases.

Chapter 8 provides an overview of the lessons learned during the research on and development of Gradoop.

Chapter 9 summarizes the dissertation and gives an outlook of open challenges and future work.

2
Background and Related Work

This chapter introduces basic concepts, terminologies, and related work as background for the following sections of this dissertation. Starting with graph data structures in Section 2.1, the main concepts of temporal graph models are given in Section 2.2. Graph streams are particular types of dynamic graphs and will be discussed in more detail in Section 2.3. An overview of temporal- and graph stream query languages is provided in Section 2.4. After a short discussion of graph databases and processing systems characteristics in Section 2.5, we finalize the chapter with an overview of temporal graph processing systems in Section 2.6.

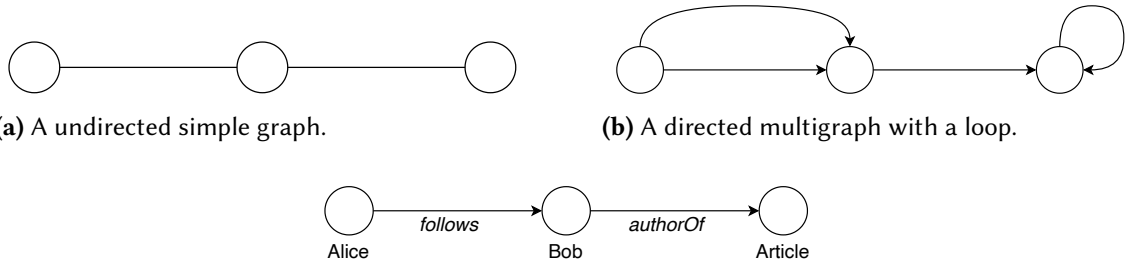
2.1 Graph Data Structures

In the data representation and analysis landscape, graph structures have established themselves as a fundamental paradigm for modeling and understanding complex relationships in various domains [189]. Graphs provide a powerful framework for capturing important connections between entities, making them ideal for modeling scenarios where relationships are as crucial as the entities themselves. This section explores the basic foundations of graph data structures to better understand more complex definitions in the subsequent chapters of this dissertation.

A graph in its simplest form is a *undirected simple graph* [56] as shown in Figure 2.1a. It consists of a set of *vertices* (or *nodes*), which represent the entities, and a set of *edges* (or *relationships*) that represent the connections among them.

Definition 1. *(Undirected Simple Graph) An undirected simple graph is a pair* $G = (V, E)$ *of distinct sets with* $E \subseteq [V]^2$*, i.e., elements of* E *are 2-element subsets of* V *[56]. The elements of* $v \in V$ *are vertices, whereas the elements of* $E \subseteq \{\{v_j, v_k\} | v_j, v_k \in V and\ v_j \neq v_k\}\}$ *are the edges of the graph, which are unordered pairs of vertices. The vertex set of a graph* G *is referred to as* $V(G)$*, its edge set as* $E(G)$*. A vertex* v *is incident with an edge* e *if* $v \in e$*. Two vertices* v_j, v_k *of* G *are adjacent (or neighbours), if* $\{v_j, v_k\}$ *is an edge of* G*.*

With a graph of this form, various scenarios can be represented, for example, friendship connections between users of a social network or railroad connections between cities. However, there are use cases where this simple structure is not sufficient. The meaning



(a) A undirected simple graph.

(b) A directed multigraph with a loop.

(c) A labeled graph.

Figure 2.1: Three different graph structures.

of a relationship may depend on its direction (e.g., a specific train goes *from* city A *to* city B), which results in the need for *directed* edges, which start at a *source vertex* and end at a *target vertex*. A graph with directed edges is called a *directed graph*. Further, a vertex may be connected with itself, which is called a *loop* and the quantity of edges between the same pair of vertices can be of importance, for example, there can be multiple train connections between the same cities, which results in a so-called *multigraph*. The definition of a simple undirected graph can be extended to a *directed multigraph with loops* [56] where Figure 2.1b shows an example. It is formally defined as follows:

Definition 2. *(Directed Multigraph) Let $G = (V, E, s, t)$ be a directed multigraph with two maps $s : E \mapsto V$ and $t : E \mapsto V$ assigning to every edge $e \in E$ a source vertex $s(e)$ and a target vertex $t(e)$ where multiple edges between the same two vertices can exist. The edges e_1 and e_2 are multiple edges, if $s(e_1) = s(e_2)$ and $t(e_1) = t(e_2)$. An edge e is a ordered pair of vertices (v_j, v_k), where $v_j = s(e)$ and $v_k = t(e)$. The set E is thus a multiset of ordered pairs $E \subseteq \{(v_j, v_k) | v_j, v_k \in V\}$. The edge e is said to be directed from $s(e)$ to $t(e)$. If $s(e) = t(e)$, the edge is called a loop.*

To add additional semantics to the vertices and edges, specific data values can be used to describe the meaning of them. The first and simplest way is adding type *labels* to vertices and edges, as shown in Figure 2.1c, where edge labels represent a follow and authorship relationship between vertices that represent two specific users and an article in a social network. Such a *directed labeled graph* is defined as follows:

Definition 3. *(Directed Labeled Graph) A directed labeled graph $G = (V, E, s, t, B, \beta)$ contains in addition to Definition 2 a set of type labels $B = \{b_1, b_2, ..., b_n\}$ and the map $\beta : (V \cup E) \to B$ assigns a single label to every vertex or edge.*

With this extension, vertices and edges can be distinguished by their respective type. To describe the vertices and edges even more precise, i.e., add further specific semantics to them, a directed labeled graph can be further extended. One possible extension is the *Property Graph Model (PGM)* [14, 170] which supports an arbitrary number of *properties* in form of key-value pairs to vertices and edges. A property graph, sometimes also called *Labeled Property Graph (LPG)*, is one of the most used graph data models in research and industry and is implemented in several open-source and commercial graph databases and processing systems [30, 88], such as Neo4j [142], JanusGraph [70] or TigerGraph [205], and processing frameworks, such as Oracle Labs PGX [93]. Figure 2.2 shows an example of a property graph, where the vertices and edges have no, one or two properties assigned besides the type label. The following Definition 4 formally defines a property graph.

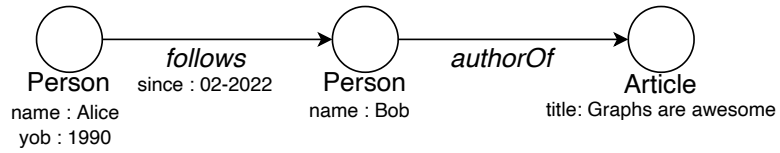


Figure 2.2: A property graph.

Definition 4. *(Property Graph) A property graph* $G = (V, E, s, t, B, \beta, K, A, \kappa)$ *contains in addition to Definition 3 a set of property keys* $K = \{k_1, k_2, ..., k_n\}$*, a set of data values* $A = \{a_1, a_2, ..., a_n\}$ *and a partial function* $\kappa : (V \cup E) \times K \rightharpoonup A$ *that assigns a data value to a combination of a vertex or edge and a property key.*

In addition to this definition, Angles allows in [14] multiple labels per vertex and edge, whereas others add a uniquely identifiable ID to them [176, 178]. With Definition 4, the property graph in Figure 2.2 can be noted as follows:

- the vertex set $V = \{v_1, v_2, v_3\}$
- the edge set $E = \{e_1, e_2\} = \{(v_1, v_2), (v_2, v_3)\}$
- the set of type labels $B = \{"Person", "Article", "follows", "authorOf"\}$
- the set of property keys $K = \{name, yob, since, title\}$
- the set of data values $A = \{"Alice", "1990", "Bob", "02 - 2022", "Graphs\ are..."\}$
- $\beta(v_1) = "Person"$, $\kappa(v_1, name) = "Alice"$, $\kappa(v_1, yob) = "1990"$
- $\beta(v_2) = "Person"$, $\kappa(v_2, name) = "Bob"$
- $\beta(v_3) = "Article"$, $\kappa(v_3, title) = "Graphs\ are\ awesome"$
- $\beta(e_1) = "follows"$, $\kappa(e_1, since) = "02 - 2022"$
- $\beta(e_2) = "authorOf"$

It is worth mentioning, that the *Resource Description Framework (RDF)* [115] is a second widely used graph data model, which is standardized and uses a triple-based structure and *International Resource Identifiers (IRI)* to represent entities and relationships. It was initially developed to describe metadata of web-resources [88]. The expression of relationships between resources is given through the utilization of subject-predicate-object triples, wherein each triple encapsulates a descriptive assertion. Subjects represent resources, predicates signify relationships, and objects denote values or associated resources.

Although both RDF and PGM are rich graph data models, they differ in their philosophical orientation and use cases. RDF embodies the ideals of the *Semantic Web* by enabling networked knowledge representation, while the PGM is primarily focused on operational and analytical use cases based on specific graph database technologies. The two data models, their suitability for particular use cases, and their interoperability have been discussed in the literature for years. Petermann [158] as well as Hofer et al. [88] provide an extensive overview of both data models as well as an interesting discussion about pros and cons for both RDF and PGM.

Knowledge Graphs (KGs) are another class of graphs that organize information to query and retrieve knowledge efficiently. It semantically represents heterogeneous entities (gene mutations, publications, patients, etc.) and their relations, typically integrated from different sources. Some KGs are represented in the PGM, like the current CovidGraph project [69], but most of them are based on RDF, like DBPedia [87], as shown by Hofer et

al. [88]. However, this dissertation orients on property graphs, but the contributions and developed concepts can be mostly adapted to RDF graphs, too.

One characteristic in all data model extensions considered in this section is particularly noticeable: the graph is *static* and does not change. We call a graph static if the sets V and E are fixed and we have no knowledge about their evolution. This is in contrast to *dynamic* or *temporal graphs* and *graph streams*, where vertices and edges change over time, and the knowledge about the evolution of the graph is part of the data model.

2.2 Temporal Graphs

Graph data models are used to model an observed network of the real-world to execute queries, algorithms and analysis on it. Such networks are rarely rigid but exhibit a high degree of dynamism, thus graph data models should natively support these dynamics. In this section, we review such dynamic or temporal graph data models in the current literature to get a basic understanding for the subsequent chapters of this dissertation.

Temporal graphs [74, 139, 176, 197, 221], i.e., graphs whose elements are unpredictably updated over time [29], occur under various pseudonyms in literature, e.g, *dynamic graphs* [226], *time-varying graphs* [38], *time-dependent graphs* [215], *evolving graphs* [5], *time-evolving graphs* [99, 100] and *temporal networks* [89, 127]. Several surveys exist about temporal graphs [38, 89, 117, 226], comparing data models, representations, algorithms and analysis.

Temporal models are common for relational databases, and support for (bi)temporal tables and time-related queries have been incorporated into the SQL standard [102, 105, 120]. By contrast, temporal *graph* models still differ in many aspects without a consensus about the most promising approach (e.g. [38, 74, 139, 197]). Such differences exist regarding the supported time semantic (valid time, transaction time or both/bitemporal), the kinds of possible changes on the graph structure and of properties, and the temporal graph representation. The latter is versatile and will thus be considered in more detail.

According to Pitoura [161], a temporal graph can either be represented as a *historical graph*[1] or a *version graph*. A historical graph is a sequence of *graph snapshots* $G = \{G_{\omega_1}, G_{\omega_2}, ...\}$, where each snapshot represent the graph state at a single instant of time ω_i. A version graph is the union of the graph snapshots in a historical graph to one single graph including all information about its evolution, i.e., each graph element (i.e., vertex, edge or attribute) is accompanied with a timestamp ω_i or time interval $\tau = [\omega_s, \omega_e)$ according to its model. A temporal graph whose graph elements are timestamped are also called *contact sequence* [89], where the name for an extension with time intervals is *interval graph* [89] or *Interval-labeled temporal graph*. Figure 2.3 and Figure 2.4 show a toy example of a temporal graph in two different representations.

The historical graph representation is given in Figure 2.3. The example models a social network with three users and an article. At the beginning of the observation, i.e., at ω_1, Bob is the author of the article and Carol follows Alice. A time instant later (at ω_2) Alice reads the article of Bob, which results in the fact that Alice follows Bob and Carol reads the article too in the next observed time instant ω_3. Newly added edges are indicated blue. One can see, that there is a graph state for each time instant that represents the

[1] Ren et al. call a sequence of graph snapshots a *graph sequence* [165].

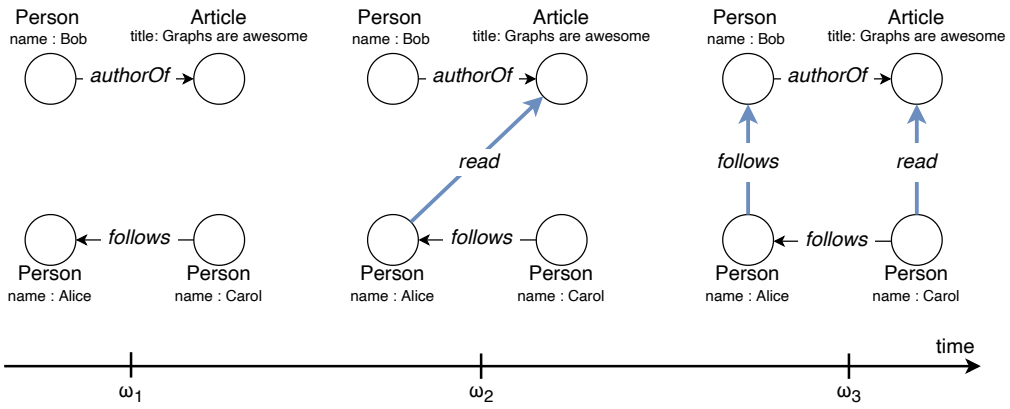


Figure 2.3: A temporal graph represented as a sequence of graph snapshots, which is called a historical graph.

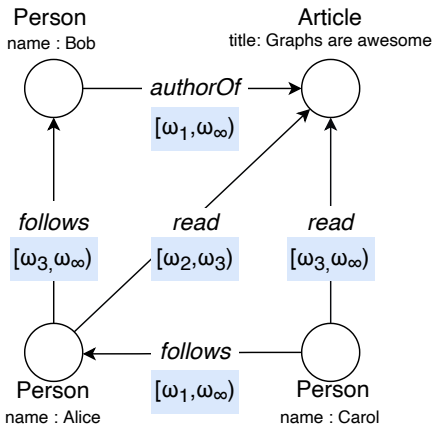


Figure 2.4: A temporal graph represented as a version graph with time intervals assigned to its edges, which is called an interval graph.

real-world at this time[2]. As Pitoura shows in [161], merging all snapshot graphs of the sequence leads to a version graph.

In Figure 2.4, the same graph is given in a version graph representation. For simplicity, just the edges were assigned with a left-close right-open time interval[3] that represents their validity. For example, the edge with label "read" between Alice and the article is just valid at ω_2, whereas the "follows" edge between Carol and Alice starts it's validity from ω_1 and do not provide information about its end, so we chose ω_∞ as a representation of a maximum timestamp.

After this brief insight into temporal graphs in general, some selected temporal property graph data models are presented in the following. There are some similarities and some differences between them, but no universally valid temporal graph model has yet been agreed upon in the literature.

Pitoura [161] defines a *temporal graph* as a tuple $G = (V, E)$, where each edge $e \in E$ is

[2]Since bitemporal modeling is already introduced in Section 1.1, the given time instants of the example could either represent the valid-time or the transaction-time domain, which is irrelevant for the given example.

[3]For a left-closed right-open time interval $\tau = [\omega_{start}, \omega_{end})$ holds: $\omega_{start} \leq \omega_i < \omega_{end}$.

a quadruple (u, v, ω_s, δ) where $u, v \in V$, ω_s is the starting time of e and δ is the delay or duration of e. It is similar to the definition of an interval graph, with the difference that the validity over a period of time is given as a timestamp and a duration. This extension to the simple undirected graph model (see Definition 1) assumes non-temporal vertices and does not define other graph characteristics like type labels or properties.

Casteigts et al. [38] introduced in their early work in 2012 so-called *time-varying graphs (TVG)*, a framework that tries to unify several concepts, formalisms and methods about temporal graphs. Since they want to cover a multitude of different temporal graph concepts, they defined a temporal graph as $G = (V, E, \mathcal{T}, \rho, \zeta, \psi, \varphi)$, where V and E represent the vertices and edges, $\mathcal{T}$ is a timespan referred to as the lifetime of the system, $\rho : E \times \mathcal{T} \rightarrow \{0, 1\}$ is a presence function indicating whether an edge is available at a given time (whereas ψ is the presence function for vertices respectively) and ζ a latency function indicating the time that it takes to traverse the edge (or φ for the vertices).

Campos et al. [34] introduced a temporal graph data model suitable for interval graphs. Time intervals are assigned to vertices only, whereas edges solely exist to model a structural relationship without any further semantic. There are attribute-vertices and value-vertices representing specific properties of a vertex. If a value changes, a new value-vertex is created and assigned to the attribute-vertex. This graph model facilitates the evolution of the graph, allowing alterations in values, attributes, objects, and relationships while retaining all data intact. With each change, a new vertex is introduced, and the concept of validity is represented through two timestamps for each vertex. This data model varies from the traditional property graph models and is reminiscent of the concepts of RDF, where properties are represented as nodes. One can see the tendency of the data model to generate very extensive graphs in relation to the network that is to be represented.

Debrouvier et al. [52] introduce a temporal graph data model, where not only nodes and relationships but also properties have a validity interval. They build their data model based on the approach of Campos et al. [34], and define a temporal property graph as $G = (N_o, N_a, N_v, E)$, where E is a set of edges, and N_o, N_a, and N_v are sets of nodes, denoted object nodes, attribute nodes, and value nodes, respectively. Further, each element is assigned with time intervals representing periods during which the element is valid. Same for value nodes, which is how they enable temporality for properties. Finally, they introduced several integrity constraints that must hold for a consistent temporal graph of their model. The disadvantage of this model is that no properties for edges are supported.

The temporal graph data model of Hartmann et al. [84] follows a different approach to extend the property graph model with temporal support. They extend vertices, edges and properties with a state that represents the beginning validity through a timestamp. The graph model is defined through a set of vertices, where each vertex maps to its incident edges and properties. Thus, edge properties are not directly supported, just by intermediate vertices like in [34] or [52].

Huang et al. [95] introduced a temporal graph model for their system TGraph with the focus on the evolution of property values. Each vertex and edge of the graph can contain static properties and dynamic properties as key-value pairs. The characteristic of the dynamic properties is that the value is an ordered list of key-value pairs, where the corresponding key represents a timestamp. This modeling is similar to a time series.

Each value, which is assigned to a timestamp, is valid from this time until the time of the following element of the list. As said before, the model just supports the evolution of properties but without supporting a structural evolution of the graph itself.

Moffitt and Stoyanovich [139] propose their approach of a temporal generalization of the property graph model called also TGraph, which captures evolution both in graph topology as well as in the properties. The main idea is to associate periods of validity with graph nodes, edges, and property values. Assuming a linearly ordered discrete time domain Ω^T with time instances $\omega \in \Omega^T$ of limited precision, a time interval that represents a discrete contiguous set of time instances from Ω^T, starting from and including the start time, continuing to but excluding the end time, they define a temporal graph (TGraph) as follows:

Definition 5. *(Temporal Property Graph according to Moffitt and Stoyanovich [139])* *A TGraph is a 6-tuple $G = (V, E, L, \rho, \xi^T, \lambda^T)$, where:*

- *V is a finite sets of vertices, E a finite set of edges, L a finite set of property labels;*
- *$\rho : E \to (V \times V)$ is a total function that maps an edge to its source and target vertex;*
- *$\xi^T : (V \cup E) \times \Omega^T \times \Omega^T \to B$ is an existence function that maps a node or edge and time period to a Boolean, indicating its existence during the time period; and*
- *$\lambda^T : (V \cup E) \times L \times \Omega^T \times \Omega^T \to val$ is a partial function that maps a node or an edge, a property label, and a time period to a value of the property during the time period.*

They further provide an alternative representation of this model using a pair of nested temporal relations and prove their equivalence [139]. The disadvantage of this model is the focus on one time dimension (unitemporal) and the lack of logical abstractions of subgraphs. However, because of the expressiveness and flexibility of the graph model presented, it will serve as one basis for the model that is developed in this dissertation (see Section 3.3 and Section 6.2).

The last evaluated temporal graph model is the one of Graphite [74], which is a system for distributed and iterative processing of temporal graphs, introduced by Gandhi and Simmhan. They propose an interval-centric computing model (ICM) together with a temporal graph data model, that covers graphs that are fully evolved and ready for processing. Assuming a linearly ordered discrete time domain Ω containing time-points ω_i and a time interval $\tau = [\omega_{start}, \omega_{end})$ with $\omega_{start}, \omega_{end} \in \Omega$ as a set of time points that are part of the interval, Gandhi and Simmhan formally define a temporal graph as follows:

Definition 6. *(Temporal Property Graph according to Gandhi and Simmhan [74])* *A temporal graph is a directed multi-graph $G = (V, E, L, A_V, A_E)$, where:*

- *V is a finite set of vertices, where each vertex $v \in V$ is a pair $\langle vid, \tau \rangle$ with vid as a unique identifier and τ as a time-interval for which the vertex is valid (its lifespan);*
- *E is a finite set of directed edges, where each edge $e \in E$ is a 4-tuple $\langle eid, vid_i, vid_j, \tau \rangle$ with eid as a unique identifier, vid_i, vid_j as the vertex identifiers of the source and target vertex and τ as the time interval for which the edge is valid (its lifespan);*
- *L is a finite set of property labels that can be associated with either vertices or edges;*
- *A_V (or A_E) is a finite set of vertex (or edge) property values, represented as a 4-tuple $\langle id, l, val, \tau_a \rangle \in A_V$ which means that a value val that is associated with a label $l \in L$ of the vertex (or edge) identified by id is valid for the interval τ_a.*

Further, a label may have distinct values for non-overlapping intervals during its vertex (or edge) lifespan.

Through the expressiveness and flexibility of this graph model, it will also serve as one basis for two models introduced in this dissertation, which are the *Temporal Property Graph Model* (TPGM) (see Section 3.3) and the TPGM+ as an extension of it (see Section 6.2). With our contribution, we provide a simple yet powerful temporal property graph model and temporal query support that avoids some of the limitations of the ones above: our model supports the modeling of subgraphs, called *logical graphs*, which are first-class entities of the model. It further supports not only single temporal property graphs but also collections of such graphs. Temporal information is modeled bitemporal, i.e., data for valid and transaction time intervals is represented within specific attributes, thereby avoiding the dedicated storage of graph snapshots (snapshots can still be determined). Further, the processing of temporal graphs is supported by temporal graph operators that can be composed within analytical programs.

A temporal graph represents the evolution of a real-world network for an observed period. If the graph-shaped data arrives continuously as a stream and queries need to be continuously evaluated concurrently with these changes, concepts from the field of stream processing are required [29]. A stream of graph data is called a *graph stream* [20, 134] or *streaming graph* [58, 155] (both terms are used interchangeably), which is introduced in the following section.

2.3 Graph Streams

A *graph stream* (also called *streaming graph*) is a special type of a dynamic graph [161], namely a graph represented in the data stream model [22]. Besta et al. provide in their survey [29] an extensive overview of the graph stream landscape including data models, representations and existing systems supporting graph stream algorithms and processing. In the following, we focus on data models for graph streams (Section 2.3.1) and the concept of windowing (Section 2.3.2.)

2.3.1 Stream Data Model

As for temporal graphs, there is no universal data model for graph streams: approaches differ in various characteristics, as visualized in the overview in Figure 2.5.

Boundedness. A graph stream can be differentiated by its boundedness. There are bounded (or finite) graph streams and unbounded (or infinite) graph streams[4]. Since the streaming model in general is an important model for the processing of massive datasets (because it is too large to fit in memory) [63], early research defined a graph stream as an alternative representation of graph with fixed size that has to be processed in the streaming model [63, 134, 214]. The most simple data model of this type is to model the edges as a sequence, sometimes with a fixed vertex set that can be stored in memory. For example, Feigenbaum et al. [63] define their interpretation of a graph

[4]The stream processing system Apache Flink also follows that differentiation: they define a fixed dataset as a *bounded stream* and a infinite stream of data as a *unbounded stream* [37].

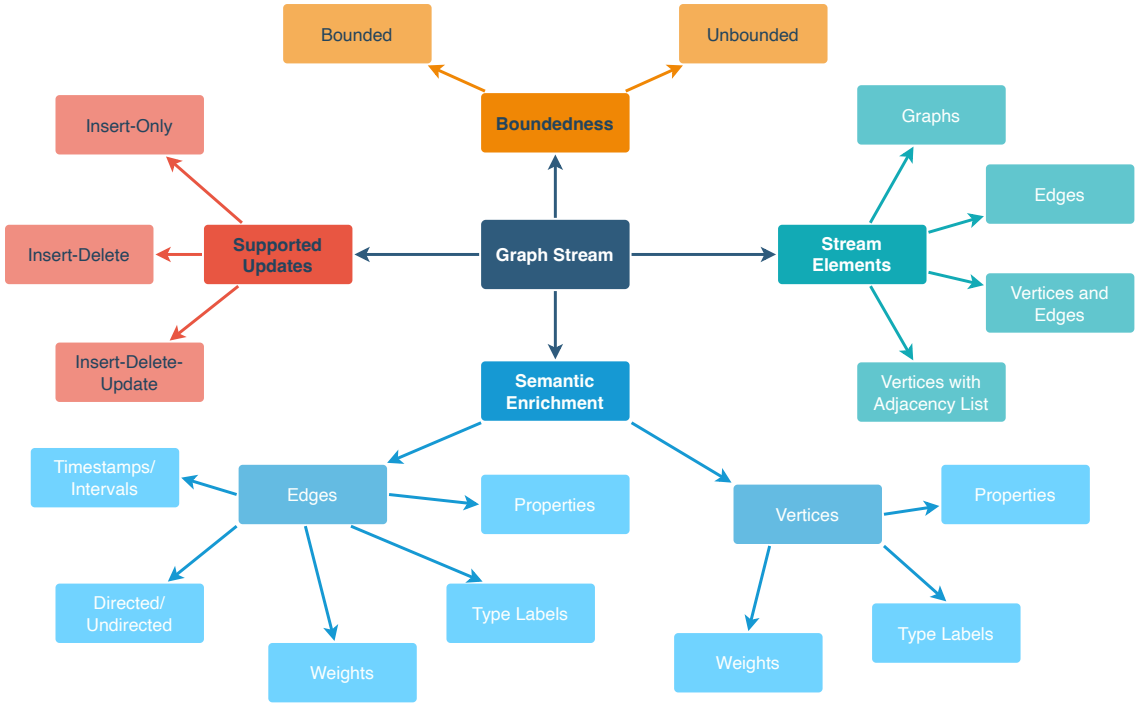


Figure 2.5: Overview of different characteristics of graph streams.

stream as a sequence of edges $e_{i_1}, e_{i_2}, \ldots, e_{i_m}$, where $e_{i_j} = (v_s, v_t) \in E$ and $i_1, i_2, \ldots, i_m$ is an arbitrary permutation of $[m] = \{1, 2, \ldots, m\}$.

However, on the other hand, a graph stream can be unbounded (infinite) [80], e.g., live user interactions on Instagram, thus modeled in the data stream model [22]. Some or all of the input data that are to be operated on arrive as one or more continuous data streams and are not available for random access from disk or memory. This paradigm leverages the evolving nature of data to capture real-time changes and interactions within complex networks, where the online analysis of such stream produce real-time results [1]. Definition 7 shows a simplified version of an unbounded graph stream defined by Pacaci et al. [155, 156].

Definition 7. *(Streaming Graph based on Pacaci et al. [156]) A streaming graph S is a constantly growing sequence of tuples $S = \{t_1, t_2, \ldots, t_m\}$ in which each tuple t_i arrives at a particular time ω_i ($\omega_i < \omega_j$ for $i < j$). A tuple t is a triple (ω, e, op) where ω is the event (application) timestamp of the tuple assigned by the data source, $e = (v_s, v_t)$ is the directed edge with source vertex v_s and target vertex v_t, and op is the type of the edge, i.e., insert (+) or delete (-).*

Stream Elements. Another distinguishing feature of graph streams is the choice of stream elements [29]. An element of the stream can be: 1) a *graph* representing an (complex) event including the contained entities and relations, e.g., two bike rental stations and a bike as vertices and the rental relationship as edges; 2) an *edge* representing the relationship between two entities that are represented by an unique identifier, e.g., follow relationships between users of a social network; 3) a *vertex and edge* in a separate stream, e.g., a stream of user vertices with meta information (name, age, etc.) and a

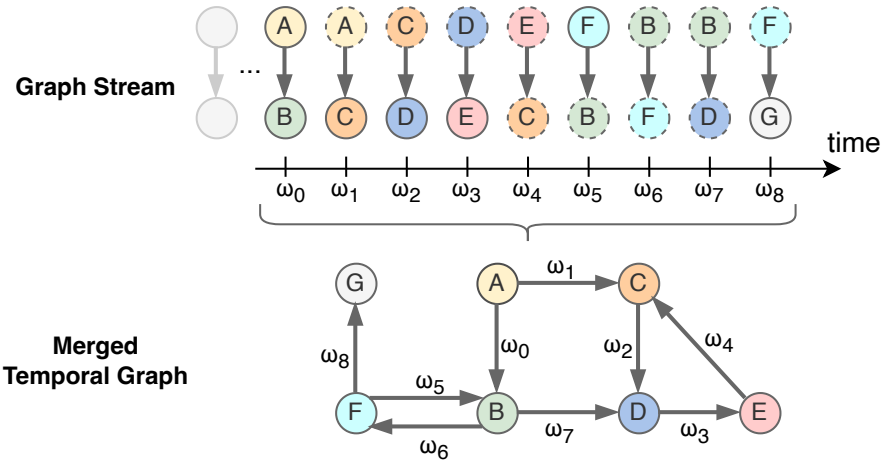


Figure 2.6: A simplified graph stream.

stream of messages as edges that are sent between the users; or 4) a *vertex* with a list of adjacent vertices, e.g., a user and its followers.

Supported Updates. If we say that a graph stream is a stream of graph updates, a graph update can be manifold. In the simplest case, the stream consists of inserts (or new elements), which is also called insert-only stream [134]. For example, an insert-only graph stream consisting of simple edges (also denoted as edge stream) can be defined as follows:

Definition 8. *(Insert-only Edge Stream) A Graph Stream* $S = \{(e_1, \omega_1), (e_2, \omega_2), \dots\}$ *is an unbounded ordered sequence of pairs* (e, ω)*, where* $e = (v_s, v_t)$ *is an edge between the source vertex* v_s *and target vertex* v_t *and* ω *is a non-decreasing timestamp.*

If the graph elements can be both added and removed, it is called an insert-delete stream. See Definition 7 for an example. There it is possible to encode in the stream that the current arriving element is a delete operation, e.g., a user does not follow another user anymore. The third variant is a graph stream that allows to update a previously seen element, e.g. to update an edge weight.

Semantic Enrichment. Unlike rich graph models like Property Graphs (see Definition 4), graph streams typically adopt simpler data models, placing their emphasis on the graph structure itself rather than extensive data associated with vertices or edges [29]. However, some models do offer basic additional data for vertices or edges, in the form of weights, type labels and/or properties. Usually edges are directed, but undirected edges are also possible. Additionally, edges can be assigned timestamps or time intervals, denoting when they were added to the stream or when the modeled relationship happened in the real-world. These time information can also indicate modifications, such as updates to the weight of an existing edge [29].

Figure 2.6 shows a simple example of an unbounded insert-only graph stream on the top and the temporal graph that results from merging the arrived elements on the bottom. For simplicity, the vertices are identified by capital letters and colors. The graph stream elements represent edge additions for each time point $\omega_1, \omega_2, \dots$ with currently ω_8 as the

most recent time in a time domain Ω. As an example, this could be a stream of *follow* relationships between users of a social network or bike *rentals* between rental stations in a micromobility network.

2.3.2 Windowing

Stream processing has emerged as the central paradigm for processing data in real time, enabling seamless analysis of constantly evolving data sets [22]. A fundamental component of stream processing is window processing, which divides the continuous stream of data into finite, manageable segments or windows. This mechanism proves to be indispensable in data stream processing as it enables operations on finite chunks of data and facilitates timely insights into dynamic information [22].

This principle can also be extended to the graph stream domain. Window processing introduces temporal boundaries into the data and allows for the examination of snapshots of the evolving graph, which is essential for tracking changes, identifying patterns, and gaining valuable insights. Just as in traditional stream processing, these windows enable efficient and meaningful operations on the evolving structure of the graph.

In addition, window processing in graph streams is related to the need to effectively manage limited computational resources. By setting temporal boundaries through a window, the allocation of resources to graph analysis can be constrained so that the computational load remains manageable [22, 29]. In this way, window processing not only enables real-time analysis of evolving graphs but also enhances the scalability and efficiency of graph stream processing systems.

According to the example graph stream in Figure 2.6, a window from ω_0 to ω_8 would create a current view on the data (also called *active* [134]) of the most recent 9 time points. If all graph elements of this window are combined, i.e. identical nodes are represented as one node, this would result in the temporal graph shown below. This view on the most recent state of the example graph stream provides the ability to apply all kinds of graph algorithms, from static to temporal ones. For example, one can find out that there are two directed paths from node A to C or the static degree of node B is with a value of 4 the highest.

In summary, the concept of window processing, well-established in stream processing, bridges to the domain of graph streams, where it plays an equal role. It empowers researchers and practitioners to follow the dynamic nature of graph data, offering a structured approach for analyzing interconnected entities while maintaining computational efficiency [29].

2.4 Query Language Extensions

Having delved into the realms of temporal graphs and graph streams, our attention now turns to the existing query languages tailored for graphs that incorporate the temporal aspect or evolution of a graph. Section 2.4.1 shows several language extensions for temporal graphs with a focus on property graphs. We refer to the survey of Analyti and Pachoulakis [11] for temporal RDF languages like T-SPARQL [81]. In addition, Section 2.4.2 shows languages supporting the continuous evaluation of graph streams.

2.4.1 Temporal Graph Query Languages

Subgraph pattern matching and navigational queries are one of the core concepts of graph queries. Having temporal graph with all information about its evolution, a query language should utilize the temporal information, e.g., to analyze different states or the evolution of the graph data. Current declarative query languages [15] for property graph databases, such as the prominent Cypher [71, 148], Gremlin [169], or Oracle PGQL [166], are powerful languages supporting mining for complex graph patterns (pattern matching), navigational expressions or aggregation queries [13, 15]. Moreover, the efforts of existing languages are converging into GQL [55, 98], the future standard graph query language that will pave the road to workload portability and shared consensus to manipulate property graphs.

Another very recent development is SQL/PGQ [55, 220], an extension to the SQL standard published in 2023 for defining and querying property graphs. However all above mentioned approaches assume static property graphs and have no built-in support for temporal queries that goes beyond the use of time and date properties. In addition, particular interval types are missing to represent a period with concrete start and end timestamps and relations between such time intervals, e.g., *precedes* or *overlaps* as defined by Allen's interval algera [10]. As a result, the analysis of graph evolution is difficult or impossible to achieve, such as determining the chronological order of a path or finding stable patterns for a particular duration of time. Similarly, queries that want to find a pattern in the graph's past or future state or that analyze the duration of relationships are complex to express without dedicated statements. Another limitation of current languages is their limited composability of graph queries, e.g., when the result of a query is a table instead of a graph.

In recent years, however, *temporal* graph query languages have been developed in research, which are mostly extensions of existing static languages.

T-Cypher [133] is part of the system Clock-G and an extension of Neo4j's Cypher with temporal constructs, such as *temporal slicing* (a temporal selection on a time point or interval), *temporal functions and operators* (temporal predicates and relations according to Allen's interval algebra [10]) and *temporal paths*. Latter are divided into 3 classes: continuous, sequential, and pairwise-continuous paths. They further provide a query processor to create a query evaluation plan.

T-GQL [52] is also based on Neo4j's Cypher and bases its naming on the currently developed GQL standard language. It provides two temporal operators, namely *Snapshot* and *Between* to query the state of the graph at a certain point in time and during the given interval, respectively. Further, the extension supports three path semantics: continuous, pairwise continuous, and consecutive path semantics. T-GQL queries are translated into (static) Cypher for query evaluation to execute them over a regular Neo4j database.

ChronoGraph [61] is an open-source versioned graph database based on Apache TinkerPop that provides temporal query extensions to the Gremlin language to support temporal graph traversals. It provides solely system-time content versioning and analysis.

In this work, we will present two additional temporal property graph languages, namely *TemporalGDL* [176], a GDL [109] and Cypher-based language to query bitemporal property graphs, and the conceptual *T-PGQL* [173], a language based on Oracle's graph query language PGQL. Former is defined in Section 3.3 whereas the latter is exemplified in Section 6.3. Both language extensions focus mainly on 3 categories of temporal extensions

that are not entirely supported in the existing ones above: 1) *bitemporal* support with complete access to time attributes of both dimensions, 2) the possibility of *projecting* the time information of the graph, including time interval bounds as well as earliest and latest time points of aggregations, 3) enabling evaluation of the whole query or specific patterns at a past or the current graph state, and 4) enable *temporal paths and patterns* through temporal constraints defined by SQL:2011 and Allen's interval algebra.

Such extensions to query languages for temporal graphs are constrained by their reliance on temporal graphs representing bounded datasets that encapsulate the historical evolution of the graph. Consequently, these extensions are unsuitable for continuous querying of graph streams, as they are tailored more for retrospective analysis rather than adapting to the dynamic nature of ongoing data streams.

2.4.2 Graph Stream Query Languages

As already discussed, *graph streams* are dynamic graphs that grow indefinitely [3], and query answering must consider a stream's unboundedness. For graph streams, the data management system is assumed to be unable to store the whole graph state, therefore it focuses on the finite sub-graph that is relevant for the query answering. Pacaci et al. introduced in [155] a data model and a query evaluation algebra on streaming graphs, including the semantics of persistent regular path queries (RPQ). Their considered stream consists of single relationships, not graphs, as in our work on Seraph, a declarative query language for graph streams, which will be introduced in Chapter 7. Our work can complement their work on streaming complex graph queries in a landscape where no declarative and industry-ready language exists.

Sakr et al. show in [189] that there is a current need for systems that can model and process both dynamic graphs and graph streams. An essential research challenge is the investigation of graph query operators for path-oriented semantics on graph streams (see Challenge C10 in Section 1.1). These are necessary for standardized graph languages like GQL. With our language Seraph and its query model and semantics, we address precisely this need. Kankanamge et al. present with Graphflow [112] a prototype in-memory graph database supporting continuous subgraph queries. Unlike Seraph, the queries cannot be evaluated on property graph streams, and windowed queries are not supported.

Our language extensions and the other existing language extensions are related to existing declarative stream processing languages, which have been around for two decades. Most of the existing solutions, including those associated with the Big Data initiative [85], present an SQL-like syntax [210], e.g. Streaming SQL [26], and build upon the Continuous Query Language model (CQL) [33]. CQL prescribes making the management of (relational) streams orthogonal to the management of relations. They defined three operator families, namely Stream-to-Relation, Relation-to-Relation, and Relation-to-Stream, which formalize the interoperability between relations and streams.

Our language Seraph follows the CQL orthogonalization principle because it makes the language compositional and maintainable [50]. However, unlike CQL, Seraph works on a stream of property graphs and provides the primitives to control the reporting completely. Learning from Dindar et al. [57], who showed how the operational semantics of stream processing engines is often uncontrollable by the user, Seraph's clauses give an end-to-end view of what impacts execution semantics from inputs to output. Hence,

Seraph users have complete control of the query execution semantics.

Finally, RSP-QL [54] was proposed by the Semantic Web community in the late 2000s' to accommodate the need for processing heterogeneous data streams. Seraph is similar to RSP-QL, which extends CQL work on RDF streams. The main differences between them are the data model, i.e., Seraph adopts streams of property graphs, and the query model: Seraph temporal approach is maintained after windowing.

2.5 Graph Database and Graph Processing Systems

Graph database systems are typically based on the PGM or RDF and provide a query language supporting operations such as pattern matching [13] and neighborhood traversal. The analysis of current graph database systems in [110] showed that they mainly focus on OLTP-like CRUD operations (create, read, update, delete) for vertices and edges as well as on queries on smaller portions of a graph, for example, to find all friends and interests of a specific user. Support for graph mining and horizontal scalability is limited since most graph database systems are either centralized or can replicate the entire database on multiple systems to improve read performance (albeit some systems also support partitioned graph storage). As already discussed for the query languages, the focus is on static graphs, so the storage and analysis of (bi)temporal graphs are not supported.

Graph processing systems are typically based on the bulk synchronous parallel (BSP) programming model [212] and provide scalability, fault tolerance, and flexibility to express arbitrary static graph algorithms. The analysis in [110] showed that they are mainly used for graph mining while they lack support for an expressive graph model such as the property graph model and a declarative query language. There is also no built-in support for temporal graphs and their analysis, so the management and use of temporal information are left to the applications.

In this thesis, we aim to combine the advantages of graph databases and graph processing systems and to provide several extensions, in particular, support for bitemporal graphs and temporal graph analysis. As mentioned, this is achieved with a new temporal property graph model TPGM and powerful graph operators that can be used within analysis programs. All operators are implemented based on Apache Flink to support parallel graph analysis on distributed cluster platforms for horizontal scalability.

2.6 Temporal Graph Processing Systems

We now discuss some selected systems for temporal graph processing that have been developed in the last decade.

Kineograph [41] is a distributed platform ingesting a stream of updates to construct a continuously changing graph. It is based on in-memory graph snapshots which are evaluated by conventional mining approaches of static graphs (e.g., community detection).

ImmortalGraph [138] (earlier known as *Chronos*) provides a storage and execution engine for temporal graphs. It is also based on a series of in-memory graph snapshots that are processed with iterative graph algorithms. Snapshots include an update log so that the graph can be reconstructed for any given point in time.

Chronograph [61] implements a dynamic graph model that accepts concurrent modifications by a stream of graph updates including deletions. Each vertex in the model has an associated log of changes of the vertex itself and its outgoing edges. Besides batch processing on graph snapshots, online approximations on the live graph are supported.

A similar system called *Raphtory* [197] maintains the graph history in-memory, which is updated through event streams and allows graph analysis through an API. The used temporal model does not support multigraphs, i.e., multiple edges between two vertices are not possible.

Tegra [100] provides ad-hoc queries on arbitrary time windows on evolving graphs, represented as a sequence of immutable snapshots. An abstraction called Timelapse comprises related snapshots, provides access to the lineage of graph elements and enables the reuse of computation results across snapshots. The underlying distributed graph store uses an object-sharing tree structure for indexing and is implemented on the GraphX API of Apache Spark. A new snapshot is created for every graph change and cached in-memory according to a least recently used approach where unused snapshots are removed from memory and written back to the file system. So unlike Gradoop, where the entire graph is kept in-memory, snapshots may have to be re-fetched from disk. Furthermore, Tegra does not provide properties on snapshot objects and focuses on ad-hoc analysis on recent snapshots while analysis with Gradoop relies more on pre-determined temporal queries and workflows.

Tink [122] is a library for analyzing temporal property graphs built on Apache Flink. Single time intervals represent temporal information for edges only, i.e., there is no temporal information for vertices or support for bitemporality. It focuses on temporal path problems and the calculation of graph measures such as temporal betweenness and closeness.

The systems *TGraph* [95] and *Graphite* [74] also use time intervals in their graph models where an interval is assigned to vertices, edges and their properties. TGraph also provides a so-called zoom functionality [5] to reduce the temporal resolution for explorative graph analysis, similar to Gradoop's grouping operator (see Section 3.3.2).

Clock-G [133] is one of the most recently published temporal graph management system. They introduce a space-efficient storage technique called δ-Copy+Log which shall optimize space usage and query evaluation time. Their temporal graph data model, the so-called Operation-based Property Graph Model (OPGM), models graph operations as actions applied on a graph entity which translates to either an addition/deletion of a vertex/edge or the update of a dynamic property [133]. The system supports three types of temporal graph queries, namely 1) point-based local queries, 2) range-based local queries, and 3) point-based and range-based global queries. Local queries are related to path traversal queries starting from a selected vertex, whereas global queries retrieve the state of a sub-graph at a time point or time range.

Compared to these systems, our work with Gradoop supports bitemporal graph data to differentiate the graph's evolution in the storage (transaction-time dimension) from the application-oriented meaning of changes (valid-time dimension). The previous systems also have less complete functionality regarding declarative graph operators and the possibility to combine them in analytical workflows for flexible graph analysis, e.g., the retrieval of a graph snapshot followed by a temporal pattern matching and a final grouping with aggregations based on the graph's evolution. Furthermore, with Gradoop

we cover a much broader range of temporal graph processing, from importing data from various sources, offering a toolbox of analytical operators up to algorithms for temporal graph metrics and the visualization of temporal graphs.

Part II

Temporal Property Graph Analysis

3
The TPGM and Gradoop

Graphs are simple yet powerful data structures to model and analyze relations between real-world data objects, whereas temporal graphs are graphs whose structure and properties change over time. The analysis of these graphs can utilize provided time information, e.g., to answer analytical questions about the graph's evolution. This section gives a complete overview of Gradoop, a graph dataflow system for scalable, distributed analytics of temporal property graphs that has been continuously developed since 2015. Its graph model TPGM allows bitemporal modeling of vertices, edges, logical graphs and graph collections. A declarative analytical language called GrALa enables analysts to flexibly define analytical graph workflows by composing different temporal operators. Built on a distributed dataflow system, large temporal graphs can be processed on a shared-nothing cluster.

The contents of this section were published under the title *Distributed Temporal Graph Analytics with Gradoop* [176].

3.1 Introduction

The analysis of graph data has gained fundamental interest, e.g., for web information systems, social networks [49], business intelligence [159, 183, 216] or in life science applications [123, 157]. There is a large spectrum of analysis forms for graph data, ranging from graph queries to find specific patterns (e.g., biological pathways), over graph mining (e.g., to rank websites or detect communities in social graphs) to machine learning on graph data, e.g., to predict new relations.

Graph datasets are often large and heterogeneous, with millions or billions of vertices and edges of different types, making the efficient implementation and execution of graph algorithms challenging [110, 187]. Furthermore, the structure and contents of graphs and

networks typically change over time, making it necessary to support temporal graph analysis instead of being limited to the analysis of static graph data snapshots. Using such time information for temporal graph queries and analysis is valuable in numerous applications and domains, e.g., to determine how an infectious disease spreads over time. Like in bitemporal databases [102, 105], time support for graph data should include both valid time (also known as application time) and transaction time (also known as system time), to differentiate *when* something has occurred or changed in the real world and *when* such changes have been recorded and thus became visible to the system.

Two major categories of systems focus on the management and analysis of graph data: graph database systems and distributed graph processing systems, as already discussed in Chapter 2. To overcome the described limitations and combine the strengths of existing systems, we started in 2015 already the development of a new open-source (`https://github.com/dbs-leipzig/gradoop`) distributed graph analysis platform called Gradoop (**Gra**ph Analytics on Ha**doop**), that has continuously been extended in the last years [78, 106, 107, 109, 111, 174, 177, 178]. Gradoop is a distributed platform to achieve high scalability and parallel graph processing. It is based on an Extended Property Graph Model (EPGM) [106] supporting both the processing of single graphs and of collections of graphs, as well as an extensible set of declarative graph operators and graph mining algorithms. Graph operators are not limited to a common query functionality such as pattern matching but also include novel operators for graph transformation or grouping. With the help of a domain-specific language called GrALa, these operators and algorithms can be easily combined within dataflow programs to implement data integration and graph analysis. While the initial focus has been on analyzing static graphs, we have recently added support for bitemporal graphs, making Gradoop a distributed platform for temporal graph analysis.

This section presents a complete system overview of Gradoop, focusing on the latest extensions for temporal property graphs. This addition required adjustments in all system components and the integration of analytical operators tailored to temporal graphs, for example, a new version of the pattern matching and grouping operators, as well as support for temporal graph queries. We also outline the implementation of these operators and evaluate their performance.

The main contributions of this section address the challenges C1, C2, C3, C4, C5, C7 and C8, defined in Section 1.1. The contributions can be summarized as follows:

- **Bitemporal graph model.** We formally outline the bitemporal property graph model TPGM used in Gradoop, supporting valid and transactional time information for evolving graphs and graph collections.
- **Temporal graph operators.** We describe the extended set of graph operators that support temporal graph analysis. In particular, we present temporal extensions of the grouping and pattern matching operator with new query language constructs to express and detect time-dependent patterns.
- **Implementation and evaluation.** We provide implementation details for the new temporal graph operators and evaluate their scalability and performance for different datasets and distributed configurations.

After a description of Gradoop's architecture in Section 3.2, the bitemporal graph data model, including an outline of all available operators and a detailed description

of selected ones, is given in Section 3.3. After explaining implementation details in Section 3.4, selected operators are evaluated in Section 3.5. We summarize our work in Section 3.6. At the end of this thesis, in Chapter 8, we discuss lessons learned, related projects, and ongoing work related to the Gradoop effort.

3.2 System Architecture Overview

With Gradoop, we provide a framework for scalable management and analytics of large, semantically expressive temporal graphs. To achieve horizontal scalability of storage and processing capacity, Gradoop runs on shared nothing clusters and utilizes existing open source frameworks for distributed data storage and processing. The difficulties of distributing data and computation are hidden beneath a graph abstraction allowing the user to focus on the problem domain.

Figure 3.1 presents an overview of the Gradoop architecture. Analytical programs are defined within our **Gr**aph **A**nalytical **La**nguage (GrALa), which is a domain-specific language for the **T**emporal **P**roperty **G**raph **M**odel (TPGM). GrALa contains operators for accessing static and temporal graphs in the underlying storage as well as for applying graph operations and analytical graph algorithms to them. Operators and algorithms are executed by the distributed execution engine, which distributes the computation across the available machines. When the computation of an analytical program is completed, results may either be written back to the storage layer or presented to the user. In the following, we briefly explain the main components, some of which are described in more detail in later sections. We will also discuss data integration support to combine different data sources into a Gradoop graph.

Distributed Storage Gradoop supports several ways to store TPGM compliant graph data. To abstract the specific storage, GrALa offers two interfaces: *DataSource* to read and *DataSink* to write graphs. An analytical program typically starts with one or more data sources and ends in at least one data sink. A few basic data sources and sinks are built into Gradoop and are always available, e.g., a file-based storage like CSV that allows

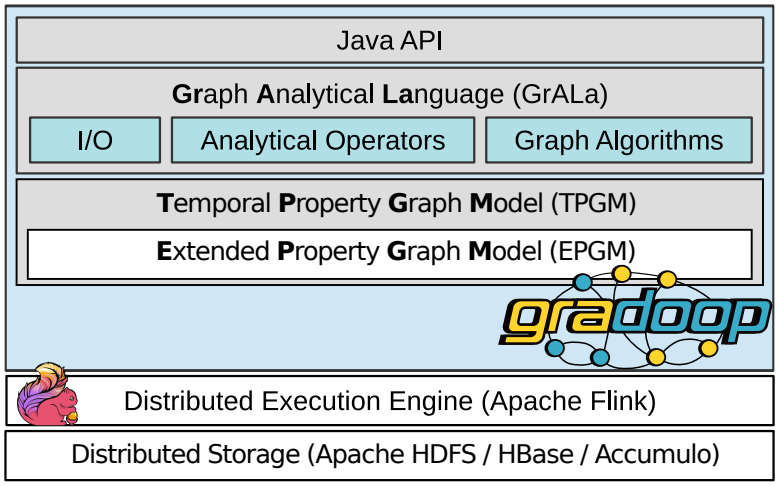


Figure 3.1: Gradoop High-Level Architecture.

reading and writing graphs from the local file system or the Apache Hadoop Distributed File System (HDFS) [192]. Gradoop in addition supports Apache HBase [65] and Apache Accumulo [67] as built-in storage, which provides database capabilities on top of the HDFS. Data distribution and replication, as well as error handling in the case of cluster failures, are handled by HDFS. A recent contribution to the open-source project [59] added *Apache Parquet* [66] and Google's data interchange format *Protobuf* [126] as a supported format for sources and sinks. Due to the available interfaces, Gradoop is not limited to the predefined storages. Other systems, e.g., a native graph database like Neo4j or a relational database, can be used as graph storage by implementing the *DataSource* and *DataSink* interfaces. Storage formats will be discussed in Section 3.4.5.

Distributed Execution Engine Within Gradoop, the TPGM and GrALa provide an abstraction for the analyst to work with graphs. However, the actual implementation of the data model and its operators are transparent to the user and hidden within the distributed execution engine. Generally, this can be an arbitrary data management system that allows implementing graph operators. Gradoop uses Apache Flink, a distributed batch and stream processing framework, that allows executing arbitrary dataflow programs in a data-parallel and distributed manner [9, 37]. Apache Flink handles data distribution along with HDFS, load balancing and failure management. From an analytical perspective, Flink provides several libraries that can be combined and integrated within a Gradoop program, e.g., for graph processing, machine learning and SQL. Apache Flink is further described in Section 3.4.1.

Temporal Property Graph Model The TPGM [178] describes how graphs and their evolution are represented in Gradoop. It is an extension of the Extended Property Graph Model (EPGM) [106] which is based on the widely accepted property graph model [14, 170]. To handle the evolution of the graph and its elements (vertices and edges), the model uses concepts of bitemporal data management [102, 105] by adding two time intervals to the graph and to each of its elements. To facilitate integration of heterogeneous data, the TPGM does not enforce any kind of schema, but the graph elements can have different type labels and attributes. The latter are exposed to the analyst and can be accessed within graph operators. For enhanced analytical expressiveness, the TPGM supports handling of multiple, possibly overlapping graphs within a single analytical program. Graphs, as well as vertices and edges, are first-class citizens of the data model and can have their own properties. Furthermore, graphs are the input and output of analytical operators which enables operator composition. Section 3.3.1 describes the TPGM model in more detail.

Graph Analytical Language Programs are specified using declarative GrALa operators. These operators can be composed as they are closed over the TPGM, i.e., take graphs as input and produce graphs. There are I/O operators to read and write graph data and analytical operators to transform or analyze graphs. Table 3.1 shows a subset of frequently used analytical operators and graph algorithms categorized by their input [106, 178]. There are specific operators for temporal graphs to determine graph snapshots or the difference between two snapshots as well as temporal versions of more general operators such as *pattern matching* [109] and *graph grouping* [111, 178]. Furthermore,

	Analytical Operators		Graph Algorithms
Temporal Graph	Aggregation	Combination	PageRank
	Pattern Matching [109]	Overlap	Community Detection
	Transformation	Exclusion	Connected Components
	Grouping [111, 178]	Equality	Single Source Shortest Path
	Subgraph	*Call**	Summarization
	Snapshot [178]		Hyperlink-Induced Topic Search
	Difference [178]		K-Means
Graph Coll.	Selection	Difference	Frequent Subgraph Mining [160]
	Distinct	Equality	
	Limit	*Apply**	
	Union	*Reduce**	
	Intersection	*Call**	

Table 3.1: Subset of frequently used analytical graph operators and algorithms available in Gradoop organized by their input type, i.e., temporal graph or graph collection. (* auxiliary operators)

there are dedicated transformation operators to support data integration [119]. Each category contains auxiliary operators, e.g., to *apply* unary graph operators on each graph in a graph collection or to *call* external algorithms. GrALa already integrates well-known graph algorithms (e.g., page rank or connected components), which can be seamlessly integrated into a program. Graph operators will be further described in Section 3.3.2.

Programming interfaces Gradoop provides two options to implement an analytical program. The most comprehensive approach is the Java API containing the TPGM abstraction including all operators defined within GrALa. Here, the analyst has the highest flexibility of interacting with other Flink and Java libraries as well as of implementing custom logic for GrALa operators. For a user-friendly visual definition of Gradoop programs and a visualization of graph results, we have incorporated Gradoop into the data pipelining tool KNIME Analytics Platform [28]. This extension makes it possible to use selected GrALa operators within KNIME analysis workflows and to execute the resulting workflows on a remote cluster [181, 182]. KNIME and the Gradoop extension offer built-in visualization capabilities that can be leveraged for customizable result and graph visualization.

Data integration support Gradoop aims at the analysis of integrated data, e.g., knowledge graphs, originating from different heterogeneous sources. This can be achieved by first translating the individual sources into a Gradoop representation and then performing data integration for the different graphs. Gradoop provides several specific data transformation operators to support this kind of data integration, e.g., to achieve similarly structured graphs (see Section 3.3.2). Furthermore, we provide extensive support for entity resolution and entity clustering within a dedicated framework called FAMER [184–186] which is based on Gradoop and Apache Flink. FAMER determines matching entities from two or more (graph) sources and clusters them together. Such clusters of matching entities can then be fused to single entities (with a *Fusion* operator) for use in an integrated

Gradoop graph. In future work, we will provide a closer integration of Gradoop and FAMER to achieve a unified data transformation and integration for heterogeneous graph data and the construction and evolution of knowledge graphs [147].

3.3 Temporal Property Graph Model

In this section, we present the Temporal Property Graph Model (TPGM) as the graph model of Gradoop that allows the representation of evolving graph data and its analysis. Having the evolutionary analysis of graphs as a focus, our goal is to model and understand the changes of the graph, provided that the historical information is available to us. We first describe the structural part of TPGM to represent temporal graph data and then discuss the graph operators as part of GrALa. The last subsection briefly discusses the graph algorithms currently available in Gradoop.

3.3.1 Graph Data Model

The Property Graph Model (PGM) [14, 170] is a widely accepted graph data model used by many graph database systems [13], e.g., JanusGraph [70], OrientDB [152], Oracle's Graph Database [47] and Neo4j [142]. A property graph is a directed, labeled and attributed multigraph. Vertex and edge semantics are expressed using *type labels* (e.g., `Person` or `knows`). Attributes have the form of key-value pairs (e.g., `name:Alice` or `classYear:2015`) and are referred to as *properties*. Properties are set at the instance level without an upfront schema definition. A *temporal* property graph is a property graph with additional time information on its vertices and edges, which primarily describes the historical development of the structure and attributes of the graph, i.e., when a graph element was available and when it was superseded. Our presented TPGM adds support for two time dimensions, valid and transaction time, to differentiate between the evolution of the graph data with respect to the real world application (valid time) and with respect to the visibility of changed graph data to the system managing the data (transaction time). This concept of maintaining two orthogonal time domains is known as bitemporality [105]. In addition, the TPGM supports graph collections, which were introduced by the EPGM [106], the non-temporal predecessor of the TPGM. A graph collection contains multiple, possibly overlapping property graphs, which are referred to as *logical graphs*. Like vertices and edges, logical graphs also have bitemporal information, a type label and an arbitrary number of properties. Before the data model is formally defined, the following preliminaries[1] have to be considered:

Preliminaries We assume two discrete linearly ordered *time domains*: Ω^{val} describes the valid-time domain whereas Ω^{tx} describes the transaction-time domain. For each domain, an instant in time is a time-point ω_i with limited precision, e.g., milliseconds. The linear ordering is defined by $\omega_i < \omega_{i+1}$, which means that ω_i happened before ω_{i+1}. A period of time is defined by a closed-open interval $\tau = [\omega_{start}, \omega_{end})$ that represents a discrete contiguous set of time instances $\{\omega | \omega \in \Omega \wedge \omega_{start} \leq \omega < \omega_{end}\}$ starting from ω_{start} and including the start time, continuing to ω_{end} but excluding the end time. To

[1] The preliminaries are partly based on model definitions of the systems TGraph [139] and Graphite [74].

separate the time intervals depending on the corresponding dimension, we use the notion τ^{val} and τ^{tx}.

Based on this, a TPGM database is formally defined as follows:

Definition 9. *(Temporal Property Graph Model database)*
A tuple $\mathbb{G} = (L, V, E, l, s, t, B, \beta, K, A, \kappa)$ *represents a temporal graph database.* L *is a finite set of logical graphs,* V *is a finite set of vertices and* E *is a finite set of directed edges with* $s : E \to V$ *and* $t : E \to V$ *assigning source and target vertex.*

Each vertex $v \in V$ *is a tuple* $\langle vid, \tau^{val}, \tau^{tx} \rangle$*, where* vid *is a unique vertex identifier,* τ^{val} *and* τ^{tx} *are the time-intervals for which the vertex is valid with respect to* Ω^{val} *or* Ω^{tx}*. Each edge* $e \in E$ *is a tuple* $\langle eid, \tau^{val}, \tau^{tx} \rangle$*, where* eid *is a unique edge identifier that allows multiple edges between the same nodes,* τ^{val} *and* τ^{tx} *are the time-intervals for which the edge exists, analogous to the vertex definition.*

B *is a set of type labels and* $\beta : L \cup V \cup E \to B$ *assigns a single label to a logical graph, vertex or edge. Similarly,* properties *are defined as sets of property keys* K*, property values* A *and a partial function* $\kappa : (L \cup V \cup E) \times K \nrightarrow A$*.*

A logical graph $G' = (V', E', \tau^{val}, \tau^{tx}) \in L$ *represents a subset of vertices* $V' \subseteq V$ *and a subset of edges* $E' \subseteq E$*.* τ^{val} *and* τ^{tx} *are the time-intervals for which the logical graph exists in the respective time dimensions. Graph containment is represented by the mapping* $l : V \cup E \to \mathbb{P}(L) \setminus \{\varnothing\}$ *such that* $\forall v \in V' : G' \in l(v)$ *and* $\forall e \in E' : s(e), t(e) \in V' \wedge G' \in l(e)$*. A graph collection* $\mathcal{G} = \{G_1, G_2, ..., G_n\} \subseteq \mathbb{P}(L)$ *is a set of logical graphs.*

Constraints Each logical graph has to be a valid directed graph, implying that for every edge in the graph, the adjacent vertices are also elements in that graph. Formally: For every logical graph $G = (V, E, \tau^{val}, \tau^{tx})$ and every edge $e = \langle eid, \tau^{val}, \tau^{tx} \rangle$ there must exist some $v_1 = \langle v_1 id, \tau_1^{val}, \tau_1^{tx} \rangle, v_2 = \langle v_2 id, \tau_2^{val}, \tau_2^{tx} \rangle \in V$ where $s(eid) = v_1 id$ and $t(eid) = v_2 id$. Additionally, the edge can only be valid with respect to Ω^{tx} when both vertices are also valid at the same time: $\tau^{tx} \subseteq \tau_1^{tx} \wedge \tau^{tx} \subseteq \tau_2^{tx}$. The same must hold for the valid time domain Ω^{val}: $\tau^{val} \subseteq \tau_1^{val} \wedge \tau^{tx} \subseteq \tau_2^{val}$.

Vertices are identified by their unique identifier and their validity in the transaction-time domain Ω^{tx}, meaning that a temporal graph database may contain two or more vertices with the same identifier but different transaction-time values. The corresponding intervals of all those vertices have to be pairwise disjoint, i.e., for every two vertices, $v_1 = \langle v_1 id, \tau_1^{val}, \tau_1^{tx} \rangle, v_2 = \langle v_2 id, \tau_2^{val}, \tau_2^{tx} \rangle \in V$ it must hold that $v_1 id = v_2 id \wedge v_1 \neq v_2 \implies \tau_1^{tx} \cap \tau_2^{tx} = \varnothing$. Edges may be identified in the same way, meaning that the graph database can also contain multiple edges with the same identifier but different transaction time values.

Figure 3.2 shows a sample temporal property graph representing a simple bike rental network inspired by the New York City's bicycle-sharing system (CitiBike) dataset [129]. This graph consists of the vertex set $V = \{v_0, .., v_4\}$ and the edge set $E = \{e_0, .., e_{10}\}$. Vertices represent rental stations denoted by corresponding type label `Station` and are further described by their properties (e.g., `name: Christ Hospital`). Edges describe the relationships or interactions between vertices and also have a type label (`Trip`) and properties. The key set K contains all property keys, for example, `bikeId`, `userType` and `capacity`, while the value set A contains all property values, for example, `21233`, `Cust` and `22`. Vertices with the same type label may have different property keys, e.g., v_1 and v_2.

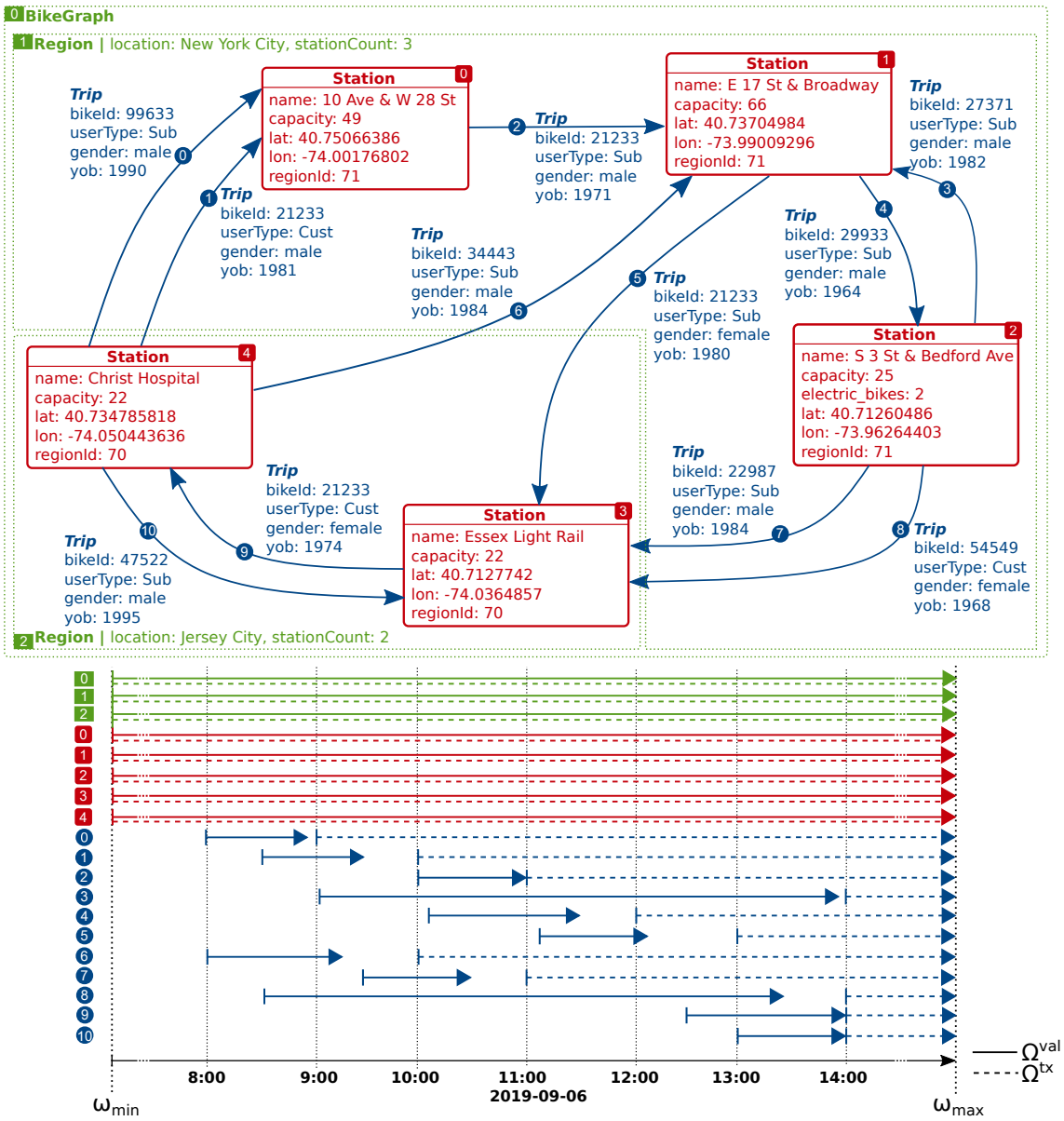


Figure 3.2: Example temporal property graph of a bike rental network.

To visualize the graph's evolution, a timeline is placed below the graph in Figure 3.2 representing the valid- and transaction-time domain Ω^{val} and Ω^{tx}, respectively. Horizontal arrows represent the validity for each graph entity and domain (dashed for Ω^{tx} and solid for Ω^{val}). One can see different time points of a selected day, as well as the minimum ω_{min} and maximum ω_{max} time as default values. The latter are used for valid times that are not given in the data record (e.g., for logical graphs or the end of a station's validity) or period ending bounds of transaction times. Edges representing a bike trip are valid according to the rental period. For example, edge e_2 represents the rental of bike 21233 at station v_0 at 10 a.m. and its return to station v_1 at 11 a.m. The bike was then rented again from 11:10 a.m. to 12:10 p.m., which is represented by edge e_5. Completed trips are stored in the graph every full hour, which is made visible by the transaction times.

The example graph consists of the set of logical graphs $L = \{G_0, G_1, G_2\}$, where G_0

represents the whole evolved rental network and the remaining graphs represent regions inside the bicycle sharing network. Each logical graph has a dedicated subset of vertices and edges, for example, $V(G_1) = \{v_0, v_1, v_2\}$ and $E(G_1) = \{e_2, e_3, e_4\}$. Considering G_1 and G_2, one can see that vertex and edge sets may not overlap since $V(G_1) \cap V(G_2) = \{\varnothing\}$ and $E(G_1) \cap E(G_2) = \{\varnothing\}$. Note that also logical graphs have type labels (e.g., `BikeGraph` or `Region`) and may have properties, which can be used to describe the graph by annotating it with specific metrics (e.g., `stationCount:3`) or general information about that graph (e.g., `location: Jersey City`). Logical graphs, like those in our example, are either declared explicitly or are the output of a graph operator or algorithm, e.g., graph pattern matching or community detection. In both cases, they can be used as input for subsequent operators and algorithms.

3.3.2 Operators

In order to express analytical problems on temporal property graphs, we defined the domain specific language GrALa containing operators for single logical graphs and graph collections. Operators may also return single logical graphs or graph collections (i.e., they are closed over the data model), thereby enabling operator composition. In the following, we use the terms *collection* and *graph collection* as well as *graph* and *logical graph* interchangeably.

Table 3.2 lists our graph operators including their corresponding pseudocode syntax for calling them in GrALa. The syntax adopts the concept of higher-order functions for several operators (e.g., to use aggregate or predicate functions as operator arguments). Based on the input of operators, we distinguish between *graph operators* and *collection operators* as well as *unary* and *binary operators* (single graph/collection vs. two graphs/collections as input). There are also *auxiliary operators* to apply graph operators on collections or to call specific graph algorithms. In addition to the listed ones, we provide operators to import external datasets to Gradoop by mapping the data to the TPGM data model, i.e., creating graphs, vertices and edges including respective labels, properties and bitemporal attributes. In the following, we focus on a subset of the operators and refer to our publications [106, 109, 111, 177, 178] and Gradoop's GitHub-Wiki [172] for detailed explanations.

Subgraph

In temporal and heterogeneous graphs, often only a specific subgraph is of interest for analytics, e.g., only persons and their relationships in a social network. The subgraph operator is used to extract the graph of interest by applying predicate functions on each element of the vertex and edge sets of the input graph. Within a predicate function, the user has access to label, properties and bitemporal attributes of the specific entity and can express arbitrary logic. Formally, given a logical graph $G(V, E)$ and the predicate functions $\varphi_v : V \to \{true, false\}$ and $\varphi_e : E \to \{true, false\}$, the subgraph operator returns a new graph $G' \subseteq G$ with $V' = \{v \mid v \in V \wedge \varphi_v(v)\}$ and $E' = \{\langle v_i, v_j\rangle \mid \langle v_i, v_j\rangle \in E \wedge \varphi_e(\langle v_i, v_j\rangle) \wedge v_i, v_j \in V'\}$.

In the following example, we extract the subgraph containing all vertices labeled `Station` having a property `capacity` with a value less than `30` and their edges of type

	Operator	GrALa	
		Input and Operator Signature	Output
Unary	Aggregation	`Graph.aggregate( propertyKey, aggregateFunction )`	Graph
	Transformation	`Graph.transform( graphFunction, vertexFunction, edgeFunction )`	Graph
	Pattern Matching	`Graph.query( patternGraph )`	Collection
	Subgraph	`Graph.subgraph( vertexPredicateFunction, edgePredicateFunction )`	Graph
	Snapshot	`Graph.snapshot( predicateFunction, [dimension] )`	Graph
	Difference	`Graph.diff( firstPredicate, secondPredicate, [dimension] )`	Graph
	Grouping	`Graph.groupBy( vertexGroupingKeys, vertexAggregateFunctions, edgeGroupingKeys, edgeAggregateFunctions )`	Graph
	Sampling	`Graph.sample( samplingAlgorithm )`	Graph
	VertexToEdge	`Graph.vertexToEdge( vertexLabel, newEdgeLabel )`	Graph
	EdgeToVertex	`Graph.edgeToVertex( edgeLabel, newVertexLabel, edgeLabelSourceToNew, edgeLabelNewToTarget )`	Graph
	PropertyToVertex	`Graph.propToVertex( vertexLabel, propertyKey, newVertexLabel, newPropertyKey, edgeDirection, edgeLabel )`	Graph
	VertexToProperty	`Graph.vertexToProp( vertexLabel, propertyKey, newPropertyKey )`	Graph
	ConnectNeighbors	`Graph.connectNeighbors( sourceVertexLabel, edgeDirection, neighborVertexLabel, newEdgeLabel )`	Graph
	Pattern Matching	`Collection.query( patternGraph )`	Collection
	Selection	`Collection.select( predicateFunction )`	Collection
	Distinct	`Collection.distinct( )`	Collection
	Limit	`Collection.limit( n )`	Collection
Binary	Equality	`Graph.equals( otherGraph, [:identity|:data] )`	Boolean
	Combination	`Graph.combine( otherGraph )`	Graph
	Exclusion	`Graph.exclude( otherGraph )`	Graph
	Overlap	`Graph.overlap( otherGraph )`	Graph
	Fusion	`Graph.fusion( otherGraph )`	Graph
	Equality	`Collection.equals( otherCollection, [:identity|:data] )`	Boolean
	Difference	`Collection.difference( otherCollection )`	Collection
	Intersect	`Collection.intersect( otherCollection )`	Collection
	Union	`Collection.union( otherCollection )`	Collection
Aux.	Apply	`Collection.apply( unaryGraphOperator )`	Graph
	Reduce	`Collection.reduce( binaryGraphOperator )`	Graph
	Call	`[Graph|Collection].callForGraph( algorithm, parameters )`	Graph
		`[Graph|Collection].callForCollection( algorithm, parameters )`	Collection
Data	Graph Head	`Graph.getGraphHead( )`	GraphHead
	Graph Heads	`Collection.getGraphHeads( [label] )`	GraphHeads
	Vertices	`[Graph|Collection].getVertices( [label] )`	Vertices
	Edges	`[Graph|Collection].getEdges( [label] )`	Edges
	Graph Transactions	`Collection.getTransactions( [label] )`	Transactions
I/O	Data source	`DataSource.readGraph( )`	Graph
		`DataSource.readCollection( )`	Collection
	Data sink	`DataSink.write( [graph|collection] )`	void

Table 3.2: TPGM graph operators specified with GrALa.

`Trip` with a property `gender` which value is equal to `female`:[2]

```
subgraph = g0
  .subgraph(
    (v => v.label == 'Station' AND v.capacity < 30),
    (e => e.label == 'Trip' AND e.gender == 'female')
  )
```

Applied to the graph G_0 of Figure 3.2, the operator returns a new logical graph described through $G' = \langle \{v_2, v_3, v_4\}, \{e_8, e_9\} \rangle$. By omitting either a vertex or an edge predicate

[2] In our listings, label and property values of an entity `n` are being accessed using dot notation, e.g., `n.label` or `n.name`.

function exclusively, the operator is also suitable to declare vertex-induced or edge-induced subgraphs respectively.

Snapshot

The snapshot operator is used to retrieve a valid snapshot of the whole temporal graph either at a specific point in time or a subgraph that is valid during a given time range. It is formally equal to the *subgraph* operator, but allows for the application of specific time-dependent predicate functions, which were partly adapted from SQL:2011 [120] and Allen's interval algebra [10].

The predicate *asOf(t)* returns the graph at a specific point in time whereas all others, like *fromTo*(t_1, t_2), *precedes*(t_1, t_2), or *overlaps*(t_1, t_2), return a graph with all changes in the specified interval. For each predicate function, the valid-time domain is used by default but can be specified through an additional argument. Note that a TPGM graph may represent the entire history of all graph changes. For analysis of the current graph state, it is therefore advisable to use the snapshot operator with the *asOf()* predicate, parameterized with the current system timestamp. Bitemporal predicates can be defined through multiple operator calls.

For example, the following GrALa operator call retrieves a snapshot of the graph for valid time 2020-09-06 at 9 a.m. and at the current system time as the transaction time:

```
1 pastGraph = g0
2   .snapshot( asOf(CURRENT_TIMESTAMP()), TRANSACTION_TIME )
3   .snapshot( asOf('2019-09-06 09:00:00'), VALID_TIME )
```

In the timeline of Figure 3.2, one can see that edges e_1, e_6, e_8 as well as all vertices and graphs meet the valid-time condition and are therefore part of the resulting graph. All visible elements exist at the current system time according to the transaction-time domain, therefore the result does not change. However, if one changes the argument of the first (transaction time) predicate to `'2019-09-06 09:55:00'`, edges e_6 and e_8 would no longer belong to the result set, since the information about these trips was not yet persisted at this point in time.

Difference

In temporal graphs, the difference of two temporal snapshots may be of interest for analytics to investigate how a graph has changed over time. To represent these changes, a difference graph can be used which is the union of both snapshots and in which each graph element is annotated as an added, deleted, or persistent element.

The difference operator of GrALa consumes two graph snapshots defined by temporal predicate functions and calculates the difference graph as a new logical graph. The annotations are stored as a property `_diff` on each graph element, whereas the value of the property will be a number indicating that an element is either equal in both snapshots (0) or added (1) or removed (-1) in the second snapshot. This resulting graph can then be used by subsequent operators to, for example, filter for *added* elements, group *removed* elements or aggregate specific values of *persistent* elements.

For the given example in Figure 3.2, the following operator call calculates the difference between the graph at 9 a.m. and 10 a.m. of the given day:

```
diffGraph = g0
  .diff(
    asOf('2019-09-06 09:00:00'),
    asOf('2019-09-06 10:00:00'),
    VALID_TIME
  )
```

The operator returns a new logical graph described through G' where $V(G') = \{v_0, \ldots, v_4\}$ and $E(G') = \{e_1, e_2, e_6, e_7, e_8\}$. Further, the property key `_diff` is added to K and the values $\{-1, 0, 1\}$ are added to A. Since all vertices and the edge e_8 are valid in both snapshots, a property `_diff:0` is added to them. The edges e_6 and e_1 are not longer available in the second snapshot, therefore they are extended by the property `_diff:-1`, whereas the edges e_2 and e_7 are annotated by `_diff:1` to show that they were created during this time period.

Time-dependent Graph Grouping

For large graphs it is often desirable to structurally group vertices and edges into a condensed graph which helps uncovering insights about hidden patterns [106, 111] and exploratory analyse an evolving graph at different levels of temporal and structural granularity. Let G' be the condensed graph of G, then each vertex in V' represents a group of vertices in V and edges in E' represent a group of edges between the vertex group members in V. Formally, $V' = \{v'_1, v'_2, ..., v'_k\}$ where v'_i is called *super vertex* and $\forall v \in V$, $s_\nu(v)$ is the super vertex of v.

Vertices are grouped together based on the values returned by *key functions*. A key function $k : V \to \mathcal{V}$ is a function mapping each vertex to a value in some set $\mathcal{V}$. Let $\{k_1, \ldots, k_n\}$ be a set of vertex grouping key functions, then $\forall u, v \in V : s_\nu(u) = s_\nu(v) \iff \bigwedge_{i=1}^{n} k_i(u) = k_i(v)$. Some key functions are provided by the system, namely *label()* $= v \mapsto \beta(v)$ mapping vertices to their label, *property(key)* $= v \mapsto \kappa(v, key)$ mapping vertices to the according property value as well as *timeStamp(...)* and *duration(...)* used to extract temporal data from elements. The latter functions can be used to extract either the start or end time of both time domains or their duration.

It is also possible to retrieve date-time fields from timestamps, like the corresponding day of the week or the month. This can be used, for example, to group edges that became valid in the same month together. Further, user defined key functions are supported by the operator, e.g., to calculate a spatial index in form of a grid cell identifier from latitude and longitude properties to group all vertices of that virtual grid cell together. The values returned by the key functions are being stored on the super vertex as new properties.

Similarly, $E' = \{e'_1, e'_2, \ldots, e'_l\}$ where e'_i is called a *super edge* and $s_\epsilon(u, v)$ is the super edge for $\langle u, v \rangle$. Edge groups are determined along the super vertices and a set of edge keys $\{k_1, \ldots, k_m\}$, where $k_j : E \to \mathcal{V}$ are grouping key functions analogous to the vertex keys, such that $\forall e, f \in E : s_\epsilon(s(e), t(e)) = s_\epsilon(s(f), t(f)) \iff s_\nu(s(e)) = s_\nu(s(f)) \wedge s_\nu(t(e)) = s_\nu(t(f)) \wedge \bigwedge_{j=1}^{m} k_j(e) = k_j(f)$. The same key functions mentioned above for vertices are also applicable for edges. Additionally, vertex and edge aggregate functions $\gamma_v : \mathcal{P}(\mathcal{V}) \to A$ and $\gamma_e : \mathcal{P}(\mathcal{E}) \to A$ are used to compute aggregated property values for grouped vertices and edges, e.g., the average duration of rentals in a group or the number of group members. The aggregate value is stored as new property at the super vertex and super edge respectively.

The following listing shows the application of the time-dependent grouping operator using GrALa:

```
1  summary = g0.groupBy(
2    // Vertex grouping key functions
3    [label(), property('regionId')],
4    // Vertex aggregate functions
5    (superVertex, vertices =>
6      superVertex['count'] = vertices.count(),
7      superVertex['lat'] = avg(vertices.lat),
8      superVertex['lon'] = avg(vertices.lon)),
9    // Edge grouping key functions
10   [label(), timeStamp(
11     VALID_TIME, FROM, HOUR_OF_DAY)],
12   // Edge aggregate functions
13   (superEdge, edges =>
14     superEdge['count'] = edges.count(),
15     superEdge['avgTripLen'] =
16       averageDuration(VALID_TIME))
```

The goal of this example is to group *Station*s and *Trip*s in the graph of Figure 3.2 by region and to calculate the number of stations and the average coordinates of stations in each region. Furthermore, we group trip edges by the hour of the day in which the trip was started and calculate the number and average duration of trips. For example, we can gain an insight into how popular each region was and which route between which regions was the most popular or took the longest all day.

In line 3 we define the vertex grouping keys. Here, we want to group vertices by type label (using the `label()` key function) and property key `regionId` (using the `property()` key function). Edges are grouped by label and by the start of the valid time interval. The `timeStamp` key function was used for the latter to extract the start of the valid time interval and to calculate the hour of the day for this time (lines 10-11). Type labels are added as grouping keys in both cases, since we want to retain this information on super vertices and edges. In lines 5-8 and 13-16, we declare the vertex and edge aggregate functions respectively. Both receive a super element (i.e., `superVertex`, `superEdge`) and a set of group members (i.e., `vertices`, `edges`) as inputs. They then calculate values for the group and attach them as properties to the super element. In our example, a `count` property is set storing the number of elements in the group. We also use the `avg` function to calculate the average value of a numeric property and the `averageDuration` function to get the average length of the valid time interval for elements. Figure 3.3 shows the resulting logical graph for this example.

Temporal Pattern Matching

A fundamental operation of graph analytics is the retrieval of subgraphs isomorphic or homomorphic[3] to a user-defined pattern graph. An important requirement in the scope of temporal graphs is the access and usage of the temporal information, i.e., time intervals and their bounds, inside the pattern. For example, given a bike-share network, an analyst

[3] GrALa support different morphism semantics, see [109].

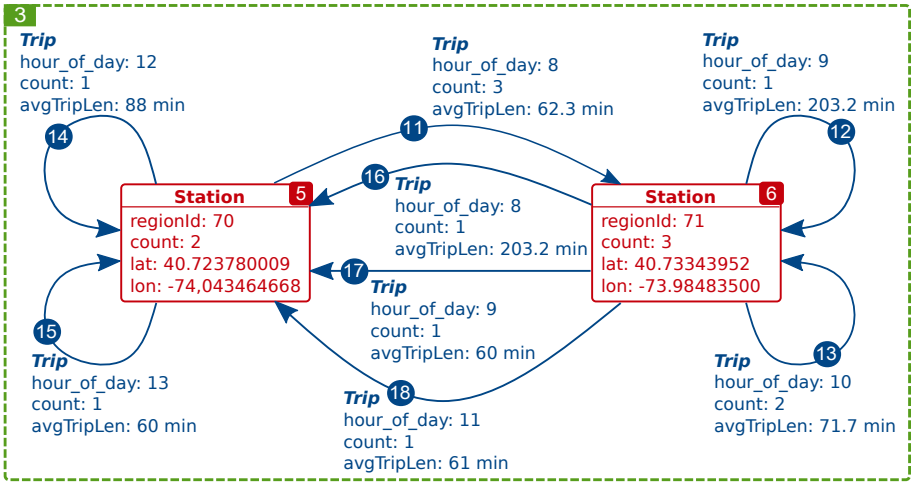


Figure 3.3: Resulting graph of the grouping example.

may be interested in a chronological sequence of trips of the same bike that started at a particular station with a radius of three hops (stations). To support such queries, GRALA provides the pattern matching operator [109], where the operator argument is a pattern (query) graph Q including predicates for its vertices and edges. To describe such query graphs, we defined *TemporalGDL*, a query language which is based on the core concepts of Cypher [71, 148], especially its `MATCH` and `WHERE` clauses. For example, the expression `(a)-[e]->(b)` denotes a directed edge `e` from vertex `a` to vertex `b` and can be used in a `MATCH` clause. Predicates are either embedded into the pattern by defining type labels and properties or expressed in the `WHERE` clause. For a more detailed description of the open-source **G**raph **D**efinition **L**anguage (GDL), the (non-temporal) language on which *TemporalGDL* is based, we refer to our previous publication [109] and the GitHub repository [108].

We extended the language by various syntactic constructs that are partly inspired by Allen's conventions [10] and the SQL:2011 standard [120], to support the TPGM-specific bitemporal attributes. These extensions enable the construction of time-dependent patterns, e.g., to define a chronological order of elements or to define boolean relations by accessing the element's temporal intervals and their bounds.

Table 3.3 gives an overview of a subset of the *TemporalGDL* syntax including access options for intervals and their bounds of both dimensions (e.g., `a.val` to get the valid time interval of the graph element that is represented by variable `a`), a set of binary relations for intervals and timestamps (e.g., `a.val.overlaps(b.val)` to check if the valid time intervals of the graph elements assigned to `a` and `b` overlap), functions to create a duration time constant of a specific time unit (e.g., `Seconds(10)` to get a duration constant of 10 seconds) and binary relations between duration constants and interval durations, e.g, `a.val.shorterThan(b.val)` to check if the duration of the valid time interval of `a` is shorter than the one of `b` or `a.val.longerThan(Minutes(5))` to check if the duration of the interval is longer than five minutes.

Pattern matching is applied to a graph G and returns a graph collection $\mathcal{G}'$, such that $G' \in \mathcal{G}' \Leftrightarrow G' \subseteq G \wedge G' \simeq Q$, i.e., $\mathcal{G}'$ contains all isomorphic (or homomorphic) subgraphs of G that match the pattern.

Syntax	Description	Return type
Creation		
`Interval(t1,t2)`	Creates an interval.	Interval
`Timestamp(t_lit)`	Creates a timestamp.	Timestamp

`Millis(num)`
`Seconds(num)`
`Minutes(num)`
`Hours(num)`
`Days(num)`
	Creates a duration time constant by the given number.	TimeConstant

Access		

`a.tx_from`
`a.tx_to`
`a.val_from`
`a.val_to`
	Access an element's interval bound of the given dimension.	Timestamp

`a.tx` `a.val`	Access an element's interval of the give dimension.	Interval
Binary relations		

`t1 {<,<=,=,!=,>=,>} t2`
`t1.before(t2)`
`t1.after(t2)`
	Simple timestamp comparisons.	Boolean

`i1.overlaps(i2)`
`i1.contains(i2)`
`i1.precedes(i2)`
`i1.succeeds(i2)`
`i1.immediatelyPrecedes(i2)`
`i1.immediatelySucceeds(i2)`
`i1.equals(i2)`
	Binary interval relations.	Boolean

`d1.longerThan(d2)`
`d1.shorterThan(d2)`
`d1.lengthAtLeast(d2)`
`d1.lengthAtMost(d2)`
	Duration comparisons.	Boolean

Legend

`a`	a variable representing a vertex/edge
`d1;d2`	a interval instance or duration time constant
`t_lit`	a timestamp literal with format: `YYYY-MM-DDTHH:MM:SS` or `YYYY-MM-DD`
`i1,i2`	a interval instance
`num`	a numerical value
`t1;t2`	a timestamp instance

Table 3.3: Overview of TemporalGDL's syntax to support temporal graph patterns.

The pattern matching operator is applied on a logical graph as follows:

```
matches = g0.query("
  MATCH (a:Station {name:'Christ Hospital'})-[e:Trip]->(b:Station)
        (b:Station)-[f:Trip]->(c:Station)
        (c:Station)-[g:Trip]->(d:Station)
  WHERE e.bikeId = f.bikeId AND f.bikeId = g.bikeId AND
        e.val.precedes(f.val) AND g.val.succeeds(f.val)")
```

The shown *TemporalGDL* pattern graph reflects the aforementioned bike-share network query. In the example, we describe a pattern of four vertices and three edges, which are assigned to variables (`a,b,c,d` for vertices and `e,f,g` for edges). Variables are optionally followed by a label (e.g., `a:Station`) and properties (e.g., `{name:'Christ Hospital'}`). More complex predicates can be expressed within the `WHERE` clause. Here, the user has access to vertex and edge properties using their variable and property keys (e.g., `e.bikeId = f.bikeId`). In addition, bitemporal information of the elements can be accessed in a similar way using predefined identifiers (e.g., `e.val`) as described before. A chronological order of the edges is defined by binary relations, for example, `e.val.precedes(f.val)`. When called for graph G_0 of Figure 3.2, the operator returns a graph collection containing a single logical graph as shown in Figure 3.4. Each graph in the result collection contains a new property storing the mapping between query variables and entity identifiers, e.g., query variable `a` is mapped to entity with id 4.

Graph transformation operators

The *transformation* operator allows simple structure-preserving in-place selection or modifications of graph, vertex and edge properties, for example, to align different sets of properties for data integration, to reduce data volume for further processing or to map property values to temporal attributes and vice versa. Transformation functions, e.g., $\nu : V \to V$ for vertices, can modify labels, properties and temporal attributes.

In addition there are several structural transformation operators to bring graph data into a desired form, e.g. for data integration with other graphs or for easier data analy-

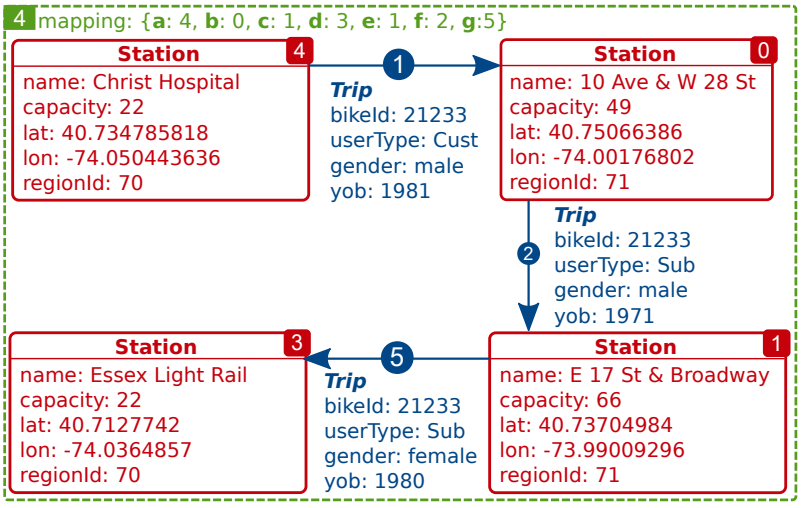


Figure 3.4: Resulting graph of Temporal Pattern Matching example.

sis [119]. For example, a bibliographic network with publications and their authors can be transformed for an easier analysis of co-authorships, e.g., by generating a graph with author vertices and co-authorship edges only. This is achieved with the help of operator *ConnectNeighbors* that creates edges between same-type vertices (e.g., authors) with a shared neighbor vertex, e.g., a co-authored publication. Further operators are available to transform properties or edges into vertices (*PropertyToVertex, EdgeToVertex*) and vice versa (*VertexToProperty, VertexToEdge*), to fuse together matching vertices, and others. A description of these operators can be can be found in [119] and Gradoop's GitHub wiki [172].

Additional graph operators

As shown in Table 3.2, GrALa provides several additional compositional graph operators.

Aggregation maps an input graph G to an output graph G' and applies the user-defined aggregate function $\alpha : L \rightarrow A$ to perform global aggregations on the graph. The output graph stores the result of the aggregate function in a new property k, such that $\kappa(G', k) = \alpha(G)$. Common examples for aggregate functions are vertex and edge count as well as more complex aggregates based on vertex and edge properties, e.g., the average trip duration in a region.

Sampling calculates a subgraph of much smaller size, which helps to simplify and speed up the analysis or visualization of large graphs. Formally, a sampling operator takes a logical graph $G(V, E)$ and returns a graph sample $G'(V', E')$ with $V' \subseteq V$ and $E' \subseteq E$. The number of elements in G' is determined by a given sample size $s \in [0, 1]$, where s defines the ratio of vertices (or edges) the graph sample contains compared to the original graph. Several sampling algorithms are implemented, three basic approaches will be briefly outlined here: *random vertex/edge sampling*, the use of neighborhood information and graph traversal techniques. The former is realized by using s as a probability for randomly selecting a subset of vertices and their corresponding edges. The same concept is applied on the edges in the random edge sampling. To improve the topological locality, *random neighborhood sampling* extends the random vertex sampling approach to include all neighbors of a selected vertex in the graph sample. Optionally, only neighbors on outgoing or incoming edges of the vertex will be taken into account. *Random walk sampling* traverses the graph along its edges, starting at one or more randomly selected vertices. Following a randomly selected outgoing edge of such a vertex, the connected neighbor is marked as visited. If a vertex has no outgoing edges, or all of them have already been traversed, the sampling jumps to another randomly selected vertex of the graph and continues traversing there. The algorithm converges when the desired number of vertices has been visited. A more detailed description of Gradoop's sampling operators can be found in [78] and the GitHub wiki [172].

Binary graph operators take two graphs as input. For example, *equality* compares two graphs based on their identifiers or their contained data, *combination* merges the vertex and edge sets of the input graphs while *overlap* preserves only those entities that are contained in both input graphs.

Graph Collection operators

Collection operators require a graph collection as input. For example, the *selection* operator filters those graphs from a collection $\mathcal{G}$ for which a user-defined predicate function $\phi : L \rightarrow \{true, false\}$ evaluates to *true*. The predicate function has access to the graph label and its properties. Predicates on graph elements can be evaluated by composing graph aggregation and selection. There are also binary operators that can be applied on two collections. Similar to graph equality, *collection equality* determines, if two collections contain the same entities or the same data. Additionally, the set-theoretical operators *union*, *intersection* and *difference* compute new collections based on graph identifiers.

It is often necessary to execute a unary graph operator on more than one graph, for example, to perform aggregation for all graphs in a collection. Not only the previously introduced operators subgraph, matching and grouping, but all other operators with single logical graphs as in- and output (i.e., $op : L \rightarrow L$) can be executed on each element of a graph collection using the *apply* operator. Similarly, in order to apply a binary operator on a graph collection, GrALa adopts the *reduce* operator as often found in functional programming languages. The operator takes a graph collection and a commutative binary graph operator (i.e., $op : L \times L \rightarrow L$) as input and folds the collection into a single graph by recursively applying the operator.

3.3.3 Iterative Graph Algorithms

In addition to the presented graph and collection operators, advanced graph analytics often requires the use of application-specific graph algorithms. One application is the extraction of subgraphs that cannot be achieved by pattern matching, e.g., the detection of communities [64] and their evolution [79].

To support external algorithms, GrALa provides generic *call* operators (see Table 3.2), which may have graphs and graph collections as input or output. Depending on the output type, we distinguish between so-called `callForGraph` and `callForCollection` operators. Using the former function, a user has access to the API and complete library of iterative graph algorithms of Apache Flink's Gelly [77], which is the Apache Flink implementation of Google Pregel [131]. By utilizing Flink's dataset iteration, co-group and flat-map functions Gelly is able to provide different kinds of iterative graph algorithms. For now, vertex-iteration, gather-sum-apply, and scatter-gather algorithms are supported. However, since Gelly is based on the property graph model we use a bidirectional translation between Gradoop's logical graph and Gelly's property graph, as described in Section 3.4.4. Thus, Gradoop already provides a set of algorithms that can be seamlessly integrated into a graph analytical program (see Table 3.1), e.g., PageRank, Label Propagation and Connected Components. Besides, we provide TPGM-tailored algorithm implementations, e.g., for frequent subgraph mining (FSM) within a graph collection [160].

3.4 Implementation

In this chapter, we will describe the implementation of the TPGM and GrALa on top of a distributed system. Since Gradoop programs model a dataflow where one or multiple temporal graphs are sequentially processed by chaining graph operators, the utilization of distributed dataflow systems such as Apache Spark [224] and Apache Flink [37] is especially promising. These systems offer, in contrast to MapReduce [51], a wider range of dataflow operators and the ability to keep data in main memory between the execution of those operators. The major challenges of implementing graph operators in these systems are identifying an appropriate graph representation and an efficient combination of the primitive dataflow operators to express graph operator logic.

As discussed in Section 2.6, the most recent approaches to large-scale graph analytics are libraries on top of such distributed dataflow frameworks, e.g., GraphX [223] on Apache Spark or Gelly [77] on Apache Flink. These libraries are well suited for executing iterative algorithms on distributed graphs in combination with general data transformation operators provided by the underlying frameworks. However, the implemented graph data models have no support for temporal graphs or collections and are generic, which means arbitrary user-defined data can be attached to vertices and edges. Consequently, model-specific operators, i.e., based on labels, properties, or time attributes, need to be user-defined, too. Hence, using those libraries to solve complex analytical problems becomes a laborious programming task.

We thus implemented Gradoop on top of Apache Flink to provide new features for flexible and general-purpose graph analytics and to benefit from existing capabilities for large-scale data and graph processing at the same time. The majority of graph algorithms listed in Table 3.1 are available in Flink Gelly. Gradoop adds automatic transformation from TPGM graphs into Gelly graphs and vice versa, as later described. In this section, we will briefly introduce Flink and its programming concepts. We will further show how the TPGM graph representation and a subset of the introduced operators, including graph algorithms, are mapped to those concepts. The last section focuses on persistent graph formats.

3.4.1 Apache Flink

Apache Flink [9, 37] supports the declarative definition and execution of distributed dataflow programs sourced from streaming and batch data. The basic abstractions of such programs are *DataSets* (or *DataStreams*) and *Transformations*. A Flink DataSet is an immutable, distributed collection of arbitrary data objects, e.g., Java Pojos or tuple types, and transformations are higher-order functions that describe the construction of new DataSets either from existing ones or from data sources. Application logic is encapsulated in user-defined functions (UDFs), which are provided as arguments to the transformations and applied to DataSet elements. Well known transformations are *map* and *reduce*, additional ones are adapted from relational algebra, e.g., *projection*, *selection*, *join*, and *grouping* (see Section 3.3.2). To describe a dataflow, a program may include multiple chained transformations. During execution Flink handles program optimization as well as data distribution and parallel processing across a cluster of machines.

The fundamental approach of sequentially applying transformations on distributed data

sets is inherited by Gradoop: Instead of generic DataSets, the user applies transformations (i.e., graph operators and algorithms) to graphs and collections of those. Transformations create new graphs which in turn can be used as input for subsequent operators hereby enabling arbitrary complex graph dataflows. Gradoop can be used standalone or in combination with any other library available in the Flink ecosystem, e.g., for machine learning (Flink ML), graph processing (Gelly) or SQL (Flink Table).

3.4.2 Graph Representation

One challenge of implementing a system for static and temporal graph analytics on a dataflow system is the design of a graph representation. Such a representation is required to support all data model features (i.e., support different entities, labels, properties and bitemporal intervals) and also needs to provide reasonable performance for all graph operators.

Gradoop utilizes three object types to represent TPGM data model elements: *graph head*, *vertex* and *edge*. A graph head represents the data, i.e., label, properties and time intervals, associated to a single logical graph. Vertices and edges not only carry data but also store their graph membership as they may be contained in multiple logical graphs. In the following, we show a simplified definition of the respective types:

```
class GraphHead{id,label,props,val,tx}
class Vertex{id,label,props,graphs,val,tx}
class Edge{id,label,sid,tid,props,graphs,val,tx}
```

Each type contains a 12-byte system managed identifier based on the UUID specification (RFC 4122, Version 1). Furthermore, each element has a label of type string, a set of properties (`props`) and two tuples (`val` and `tx`) representing time intervals of the valid- and transaction-time dimension. Each tuple consists of two timestamps that define the interval bounds. Each timestamp is a 8-byte long value that stores Unix-epoch milliseconds. Since TPGM elements are self-descriptive, properties are represented by a key-value map whereas the property key is of type string and the property value is encoded in a byte array. The current implementation supports values of all primitive Java types as well as arrays, sets and maps of those. Vertices and edges maintain their graph membership in a dedicated set of graph identifiers (`graphs`). Edges additionally store the identifiers of their incident vertices (i.e., `sid`/`tid`).

Programming abstractions

Graph heads, vertices and edges are exposed to the user through two main programming abstractions: *LogicalGraph* and *GraphCollection*. These abstractions declare methods to access the underlying data and to execute GrALa operators. Table 3.2 contains an overview of all available methods including those for accessing graph and graph collection elements as well as to read and write graphs and graph collections from and to data sources and data sinks.

The following example program demonstrates the basic usage of the Java API.

```
TemporalGraph graph = new TemporalCSVDataSource(⋯)
  .getTemporalGraph();

```

```
4  TemporalGraphCollection triangles = graph
5    .snapshot(new Overlaps('2019-09-06', '2019-09-07'))
6    .subgraph((e => e.yob > 1980))
7    .callForGraph(new PageRank('pr', 0.8, 10))
8    .query("
9      MATCH (p1)-->(p2)-->(p3)<--(p1)
10     WHERE ((p1.pr + p2.pr + p3.pr) / 3) > 0.8)
11   ");
12
13 new TemporalCSVDataSink(…).write(triangles);
```

We start by reading a logical graph from a specific data source (here in CSV format) in line 2. We then retrieve a snapshot with all elements that overlap the given period in the past in line 5. After that, in line 6, we extract an edge-induced subgraph containing only edges with a property yob that is greater than the value 1980 and all source and target vertices. Based on that subgraph, we call the PageRank algorithm (line 7) and store the resulting rank as a new vertex property pr. Using the match operator (lines 9 and 10), we extract triangles of vertices in which the total page rank exceeds a given value. The resulting collection of matching triangles is stored using a specific data sink (line 13). Note that the program is executed lazily by either writing to a data sink or by executing specific action methods on the underlying DataSets, e.g., for collecting, printing, or counting their elements.

In a second scenario, related to the bike-sharing example of Section 3.3, we want to answer the following question: *In 2019, how did the minimum, maximum and average trip duration change per month for male and female users born after 1990 between stations located in different areas?*

The following code snippet exemplifies the workflow to answer this question.

```
1  TemporalGraph summary = bikeGraph
2    // get all elements that overlap year 2019
3    .snapshot(new Overlaps('2019-01-01', '2019-12-31'))
4    // filter edges by year of birth
5    .subgraph((e => e.yob > 1990))
6    // summarize the graph
7    .groupBy(
8      // group vertices by label and grid id
9      [
10       label(),
11       v -> getGridId(v)
12     ],
13     // do not aggregate vertices
14     [ ],
15     // group edges by label, month and gender property
16     [
17       label(),
18       timeStamp(VALID_TIME, FROM, ChronoField.MONTH_OF_YEAR),
19       property("gender")
20     ],
21     // calc min, max and avg duration for grouped edges
22     [
```

```
23        new MinDuration("minDur", VALID_TIME),
24        new MaxDuration("maxDur", VALID_TIME),
25        new AvgDuration("avgDur", VALID_TIME)
26      ]
27    );
28
29  new TemporalCSVDataSink(…).write(summary);
```

We first use the *snapshot* operator (line 3) to retrieve all information about trips, users, and stations of the whole year of 2019. We further apply a *subgraph* operator with specific predicates (line 5) to filter for users born after 1990. At the end of our pipeline, we want to summarize our graph by calling the *grouping* operator (`groupBy`) with specific grouping key functions for vertices (line 10 and line 11) and edges (line 17-19). Besides the predefined *label()* function, we also show the usage of a user-defined grouping key function (line 11). It calculates a map grid using latitude and longitude properties of our vertices. We further group the edges for each month of the year, separated by the two genders of the users. During this step, we also apply multiple aggregation functions to calculate the minimum, maximum, and average duration of trips (line 23-25).

The example illustrates the level of abstraction using our operators. A user does not have to care about the underlying graph data structure, operator implementation, or distributed execution details. In Section 3.5, we will provide more complex examples as part of our evaluation.

Graph Layouts

While the two programming abstractions provide implementation-independent access to the GrALa API, their internal Flink DataSet representations are encapsulated by specific *graph layouts*. The most common *GVE Layout* (**G**raph-**V**ertex-**E**dge layout) is the default for single logical graphs and graph collections. The layout corresponds to a relational view of a graph collection by managing a dedicated Flink DataSet for each TPGM element type and using entity identifiers as primary and foreign keys. Operations that combine data, e.g., computing the outgoing edges for each vertex, require join operations between the respective DataSets. Since graph containment information is embedded into vertex and edge entities, an additional DataSet storing mapping information is unnecessary. Another experimental layout is *Indexed GVE*, a variation of the GVE layout in which vertex and edge data are partitioned into separate DataSets based on the entity label. Other user-defined graph layouts can be easily integrated by implementing against a provided interface.

Figure 3.5 shows an example instance of the GVE layout for a graph collection containing logical graphs of Figure 3.2. The first DataSet L stores the data attached to logical graphs, vertex data is stored in a second DataSet V and edge data is in a third, E. Vertices and edges store a set of graph identifiers which is a superset of the graph identifiers in L as an entity can be contained in additional logical graphs (e.g., G_0 and G_1). A logical graph is a special case of a graph collection in which the L DataSet contains a single element. Each element stores in addition the two time intervals to capture the visibility for the valid- and transaction-time dimension.

DataSet<GraphHead>

id	label	properties	tx	val
0	BikeGraph	{}	[2019-05-21, 9999-12-31)	[2013-06-01, 9999-12-31)
1	Region	{location: New York City, ...}	[2019-05-21, 9999-12-31)	[2013-06-01, 9999-12-31)
2	Region	{location: Jersey City, ...}	[2019-05-21, 9999-12-31)	[2013-06-01, 9999-12-31)

DataSet<Vertex>

id	label	properties	graphs	tx	val
0	Station	{name: 10 Ave & W 28 St, ...}	[0,1]	[2019-05-21, 9999-12-31)	[2013-06-01, 9999-12-31)
1	Station	{name: E17 St & Broadway, ...}	[0,1]	[2019-05-21, 9999-12-31)	[2013-06-01, 9999-12-31)
2	Station	{name: S 3 St & B. Ave, ...}	[0,1]	[2019-05-21, 9999-12-31)	[2013-06-01, 9999-12-31)
3	Station	{name: Essex Light Rail, ...}	[0,2]	[2019-05-21, 9999-12-31)	[2015-11-10, 9999-12-31)
4	Station	{name: Christ Hospital, ...}	[0,2]	[2019-05-21, 9999-12-31)	[2015-11-03, 9999-12-31)

DataSet<Edge>

id	label	srcId	trgId	properties	graphs	tx	val
0	Trip	4	0	{bikeId: 99633, ...}	[0]	[2019-05-21, 9999-12-31)	[2019-09-06, 2019-09-06
1	Trip	4	0	{bikeId: 21233, ...}	[0]	[2019-05-21, 9999-12-31)	[2019-09-06, 2019-09-06)
2	Trip	0	1	{bikeId: 21233, ...}	[0,1]	[2019-05-21, 9999-12-31)	[2019-09-06, 2019-09-06)
				...			

Figure 3.5: GVE layout of Gradoop. The accuracy of the timestamps has been reduced for readability reasons.

3.4.3 Graph Operators

The second challenge that needs to be solved when implementing a graph framework on a dataflow system is the efficient mapping of graph operators to transformations provided by the underlying system. Table 3.4 introduces a subset of transformations available in Apache Flink. Well-known transformations have been adopted from the MapReduce paradigm [51]. For example, the *map* transformation is applied on a DataSet of elements of type `IN` and produces a DataSet containing elements of type `OUT`. Application-specific logic is expressed through a user-defined function (`udf: IN -> OUT`) that maps an element of the input DataSet to exactly one element of the output DataSet. Further DataSet transformations are well-known from relational systems, e.g., *select* (*filter*), *join*, *group-by*, *project* and *distinct*.

Subsequently, we will explain the mapping of graph operators to Flink transformations. We will focus on the operators introduced in Section 3.3: Subgraph, Snapshot, Difference, Time-dependent Grouping, and Temporal Pattern Matching. For all operators, we assume the input graph to be represented in the GVE layout (see Section 3.4.2).

Subgraph The subgraph operator takes a logical graph and two user-defined predicate functions (one for vertices, one for edges) as input. The result is a new logical graph containing only those vertices and edges that fulfill the predicates. Figure 3.6 illustrates the corresponding dataflow program. The dataflow is organized from left to right, starting from the vertex and edge DataSets of the input graph. Descriptions on the arrows highlight the applied Flink transformation and its semantics in the operator context. First, we use the *filter* transformation to apply the user-defined predicate functions on the vertex and edge DataSets (e.g., `(v => v.capacity >= 40)`). The resulting vertex DataSet V_1 can already be used to construct the output graph. However, we have to ensure, that no dangling edges exist, i.e., only those filtered edges are selected where source and target vertex are contained in the output vertex set. To achieve that, the operator performs a *join* transformation between filtered edges and filtered vertices for both, `sourceId`

Name	Description
Map	The map transformation applies a user-defined map function to each element of the input DataSet. Since the function returns exactly one element, it guarantees a one-to-one relation between the two DataSets. `DataSet<IN>.map(udf: IN -> OUT) : DataSet<OUT>`
FlatMap	The flatMap transformation applies a user-defined flat-map function to each element of the input DataSet. This variant of a map function can return zero, one or arbitrary many result elements for each input element. `DataSet<IN>.flatMap(udf: IN -> OUT) : DataSet<OUT>`
Filter	The filter transformation evaluates a user-defined predicate function to each element of the input DataSet. If the function evaluates to `true`, the particular element will be contained in the output DataSet. `DataSet<IN>.filter(udf: IN -> Boolean) : DataSet<IN>`

Project	The projection transformation takes a DataSet containing a tuple type as input and forwards a subset of user-defined tuple fields to the output DataSet.
	`DataSet<TupleX>.project(fields) : DataSet<TupleY>`
	`(X,Y in [1,25])`

Equi-Join	The join transformation creates pairs of elements from two input DataSets which have equal values on defined keys (e.g., field positions in a tuple). A user-defined join function is executed for each of these pairs and produces exactly one output element.
	`DataSet<L>.join(DataSet<R>)`
	`.where(leftKeys)`
	`.equalTo(rightKeys)`
	`.with(udf: (L,R) -> OUT) : DataSet<OUT>`

ReduceGroup	DataSet elements can be grouped using custom keys (similar to join keys). The ReduceGroup transformation applies a user-defined function to each group of elements and produces an arbitrary number of output elements.
	`DataSet<IN>.groupBy(keys)`
	`.reduceGroup(udf: IN[] -> OUT[]) : DataSet<OUT>`

Table 3.4: Subset of Apache Flink DataSet transformations. We define `DataSet<T>` as a DataSet that contains elements of type `T` (e.g., `DataSet<String>`, `DataSet<Vertex>` or `DataSet<Tuple2<Int,Int>>`).

and `targetId` of an edge. Edges that have a join partner for both transformations are forwarded to the output DataSet. During construction of the output graph, a new graph head is generated and used to update graph membership of vertices and edges.[4]

Snapshot The snapshot operator [178] provides the retrieval of a valid snapshot of the entire temporal graph by applying a temporal predicate, e.g., `asOf` or `fromTo`. We implemented the operator analogous to the subgraph operator by using two Flink *filter*

[4]Depending on the size of the filtered DataSets, the transformations need to be distributed which might require data shuffling. Since the `join` step is used for verification, it also can be disabled if domain knowledge allows it.

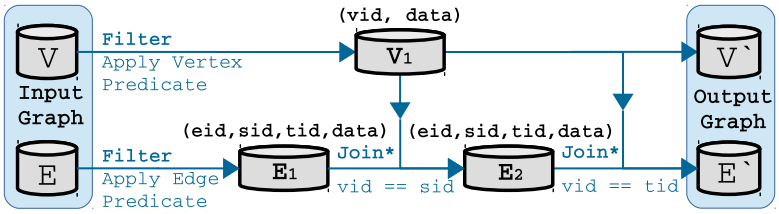


Figure 3.6: Dataflow implementation of the Subgraph operator using Flink DataSets and transformations.

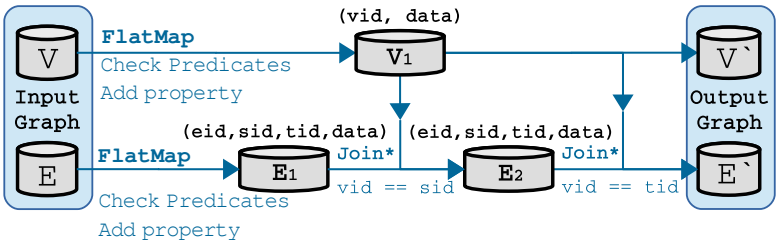


Figure 3.7: Dataflow implementation of the Difference operator using Flink DataSets and transformations.

transformations. Each transformation applies a temporal predicate to each record of V and E, respectively. Remember that a graph in TPGM is fully evolved and contains the whole history of all changes for both time-dimensions. Therefore, the predicate will check the valid or transaction time of each graph element, depending on an optional identifier of the dimension to be used, and thus decides whether to keep the element or to discard it. Just like subgraph, the filter may produce dangling edges, since vertices and edges are handled separately. The subsequent verification step (two `join` transformations) is thus performed to remove these dangling edges.

Difference To explore the changes in a graph between two snapshots, i.e., states of the graph at a specific time, we provide the difference operator (see Section 3.3.2), which we previously introduced [178]. In our case, a snapshot is represented by the same predicates that can be used within snapshot operator. The operator is applied on a logical graph and produces a new logical graph that contains the union of elements of both snapshots, where each element is extended by a property that characterizes it as added, deleted or persistent.

The architectural sketch of the difference operator is shown in Figure 3.7. Again, since all temporal information is stored in the input graph, we can apply the two predicates on the same input dataset, i.e., V or E, respectively. This gives us an advantage, as each element only has to be processed once in a single `flatMap` transformation, which has a positive effect on distributed processing. Now we can check if that element exists in both, none, only the first or only the second snapshot. It will be collected and annotated with a property as defined in Section 3.3.2, or discarded if it does not exist in at least one snapshot. The annotation step is also implemented inside the `flatMap` function. The resulting set of (annotated) vertices and edges is thus the union of the vertices and edges of both logical snapshots. Dangling edges are removed analogous to subgraph and snapshot by two *joins*.

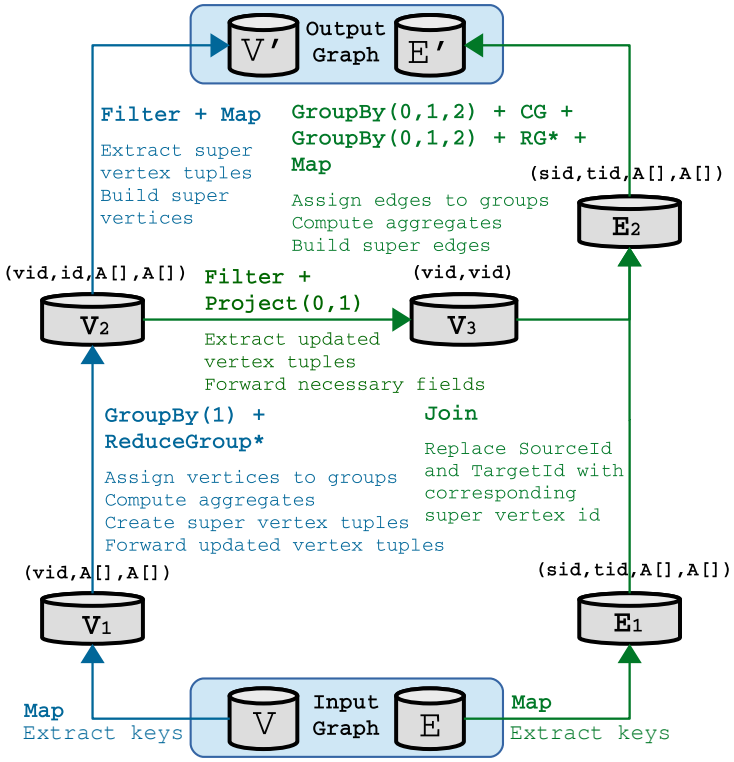


Figure 3.8: Dataflow implementation of the grouping operator using Flink DataSets and transformations. Lists of property values are denoted by `A[]`.

Time-dependent Graph Grouping The grouping operator is applied on a single logical graph and produces a new logical graph in which each vertex and edge represents a group of vertices and edges of the input graph. The algorithmic idea is to group vertices based on values returned by *grouping key functions* (or just *key functions*). Elements for which every one of these functions returns the same value are being grouped together. The group is then represented as a so called *super vertex* and a mapping from vertices to super vertices is extracted. Edges are additionally grouped with their source and target vertex identifier. Figure 3.8 shows the corresponding dataflow program.

Given a list of vertex grouping key functions, we start by mapping each vertex $v \in V$ to a tuple representation containing the vertex identifier, values returned by each of the key functions and property values needed for the declared aggregation functions (DataSet V_1). In the second step, these vertex tuples are grouped on the previously determined key function values (position 1 in the tuple). Each group is then processed by a *ReduceGroup* function with two main tasks: (1) creating a *super vertex tuple* for each group and (2) creating a mapping from vertices to super vertices via their identifier. The super vertex tuple has a similar structure to the vertex tuple, except that it stores the super vertex identifier, the grouping keys and calculated aggregate values for every aggregation function. In the final step for vertices, we construct super vertices from their previously calculated tuple representation. We therefore filter out those tuples from the intermediate DataSet V_2 and apply a map transformation to construct new `Vertex` instances for each tuple.

After applying another filter to DataSet V_2 we get a mapping from vertex to super vertex identifier (DataSet V_3) which can in turn be used to update the input edges in

DataSet E and to group them to get super edges E'. Similar to the first step for vertices, DataSet E_1 stores a representation of edges as tuples, including their source and target identifier, values of grouping key functions and property values used for aggregation functions. We can then join this DataSet with V_3 twice, first to replace each source identifier with the identifier of the corresponding super vertex and again to replace the target identifier. Since the resulting edges (now stored in DataSet E_2) are logically connecting super vertices, we can group them on source and target identifier as well as key function values. This step yields tuple representations of super edges, which finally mapped to the new `Edge` instances, representing the final super edges.

Similar to the other operators, the resulting vertex and edge DataSets are used as parameters to instantiate a new logical graph G' including a new graph head and updated graph containment information.

Temporal Graph Pattern Matching The graph pattern matching operator takes a single logical graph and a Cypher-like pattern query as input and produces a graph collection where each contained graph is a subgraph of the input graph that matches the pattern. As Flink already provides relational dataset transformations, our approach is to translate a query into a relational operator tree [91, 109] and eventually in a sequence of Flink transformations. For example, the label and property predicates within the query

```
MATCH (a:Station) WHERE a.capacity = 25
```

are transformed into a selection with two conditions $\sigma_{label='Station' \wedge capacity=25}(V)$ and evaluated using a filter transformation on the vertex DataSet. Structural patterns are being decomposed into join operations. For example the query

```
MATCH (a)-->(b)
```

is transformed into two join operations on the vertex and edge DataSets, i.e., $V \bowtie_{id=sid} E \bowtie_{tid=id} V$.

Figure 3.9 shows a simplified[5] dataflow program for the following temporal query:

```
1 MATCH (a:Station)<-[e:Trip]-(c:Station)
2       (b:Station)<-[f]-(c)
3 WHERE e.val.asOf(Timestamp(2019-09-06)) AND
4       e.val.overlaps(f.val) AND
5       a.capacity > b.capacity
```

We start by filtering vertices and edges that are required to compute the query result. Predicates are evaluated as early as possible. Especially when specifying temporal predicates with constant time values the amount of data is often enormously reduced by the filtering. In addition, only property values needed for further predicate evaluation are being kept. For example, to evaluate `(e:Trip)` and `e.val.asOf(...)`), we introduce a *Filter* transformation on the input vertex DataSet E_0 and a subsequent *Map* transformation to convert the edge object into a tuple containing only the edge id, source id and target id for subsequent joins and the `val` interval (that represents the edge's

[5]Several steps are being simplified for clarifying the operator logic. For example, the filter and map transformations are actually implemented using a single flatMap transformation to avoid unnecessary serialization. Morphism checks are being executed in a flatJoin transformation that allows to implement a filter on each join pair in a single UDF.

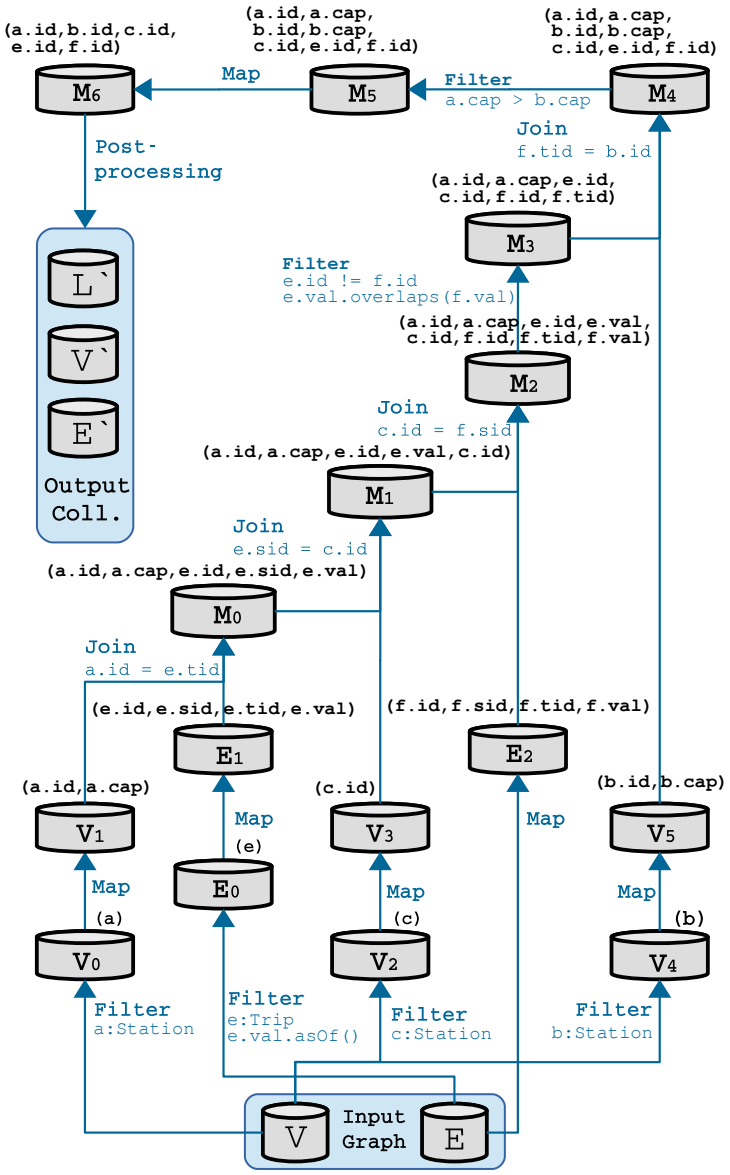


Figure 3.9: Dataflow representation of a pattern matching query.

valid-time) for later predicate evaluation (DataSet E_1). *Join* transformations compute partial structural matches (DataSet M_i) and subsequent *Filter* transformations validate the edge isomorphism semantics that demands a *Filter* transformation on DataSet M_2. Predicates that span multiple query variables (e.g., `a.capacity > b.capacity`) can be evaluated as soon as the required input values are contained in the partial match (DataSet M_4). Each row in the resulting DataSet represents a mapping between the query variables and the data entities and is converted into a logical graph in a post-processing step.

Since there is generally a huge number of possible execution plans for a single query, Gradoop supports a flexible integration of query planners to optimize the operator order [109]. Our reference implementation follows a greedy approach which iteratively constructs a bushy query plan by estimating the output cardinality of joined partial matches and picking the query plan with minimum cost. Cardinality estimation is based on statistics about the input graph (e.g., label and property/id distributions). Predicate

evaluations and property projections are generally planned as early as possible.

3.4.4 Iterative Graph Algorithms

Gelly, the Graph API of Apache Flink, uses iteration operators to support large-scale iterative graph processing [77]. It provides a library of graph algorithms as well as methods to create, transform and modify graphs. In Gradoop, iterative graph algorithms, e.g., PageRank, Connected Components, or Triangle counting, are implemented by using this Graph API. We provide a base class called `GradoopGellyAlgorithm` which includes transformation functions to translate a logical graph to a Gelly graph representation.

Various algorithms are already available in Gradoop and can be integrated into an analytical GrALa pipeline using the *Call* operator (see auxiliary operators in Table 3.2). Custom or other Gelly algorithms can also be integrated simply by extending the aforementioned base class and using the provided graph transformations. Algorithm results, e.g., the PageRank scores of vertices, the component identifiers, or the number of triangles, are typically integrated in the resulting logical graph by adding new properties to the graph (head), vertices, or edges.

3.4.5 Graph Storage

Following the principles of Apache Flink, Gradoop programs always start with at least one data source, end in one or more data sinks and are lazily evaluated, i.e., program execution needs to be triggered either explicitly or by an action, such as counting DataSet elements or collecting them in a Java collection. Lazy evaluation allows Flink to optimize the underlying dataflow program before it is being executed [37]. To allow writing to multiple sinks within a single job, a Gradoop data sink does not trigger program execution.

To define a common API for implementers and to easily exchange implementations, Gradoop provides two interfaces: `DataSource` and `DataSink` with methods to read and write logical graphs and graph collections, respectively (see the listing in Section 3.4.2). Notwithstanding, Gradoop contains a set of embedded storage implementations, including file formats and NoSQL stores.

We provide several formats to store graphs and graph collection within files. A prominent example is the CSV format which stores data optimized for the GVE layout (see Section 3.4.2). A separate metadata file contains information about labels, property keys and property value types. Generally, a sensible approach is to store the graph in HDFS using the CSV format, run analytical programs and export the result using the CSV data sink for simpler post-processing.

In addition to the file-based formats, Gradoop supports two distributed database systems for storing logical graphs and graph collections: HBase [65] and Accumulo [67]. Both storage engines follow the BigTable approach allowing wide tables including column families and fast row lookup by primary keys [40].

Another supported format which is purely used for visualization purpose is the open graph description format DOT [60, 75]. A detailed description of both sources and sinks can be found in Gradoop's GitHub wiki [172].

3.5 Evaluation

This section is split into two parts. First we show scalability results for a subset of individual TPGM operators, namely *Snapshot*, *Difference*, *Time-dependent Grouping* and *Temporal Pattern Matching*. In the second part, we analyze the performance evaluation for an analytical program composing several operators. In both cases we evaluate runtimes and horizontal scalability with respect to increasing data volume and cluster size. The experiments have been run on a cluster with 16 worker nodes. Each worker consists of a E5-2430 6(12) 2.5 Ghz CPU, 48 GB RAM, two 4 TB SATA disks and runs openSuse 13.2, Hadoop 2.7.3 and Flink 1.9.0. On a worker node, a Flink Task Manager is configured with 6 task slots and 40GB memory. The workers are connected via 1 Gigabit Ethernet.

We use two datasets referred to as *LDBC* and *CitiBike* in the evaluation. The LDBC dataset generator creates heterogenous social network graphs with a fixed schema and structural characteristics similar to real-world social networks, e.g. dynamic updates [218] and p-law distribution [97]. *CitiBike* is a real-world dataset describing New York City bike rentals since 2013 [129]. The schema of this dataset corresponds to the one in Figure 3.2 in Section 3.3. Table 3.5 contains some statistics about the two datasets considering different scaling factors (SF) for LDBC. The largest graph (SF=100) has about 283 million vertices and 1.8 billion edges. The *CitiBike* graph covers data over almost eight years with up to 20M new edges per year.

To evaluate individual operators, we execute each workflow as follows: First we read a graph from a HDFS data source, execute the specific operator and finally write all results back to the distributed file system. The graph analytical program is more complex and used to answer the following analytical questions: *What are the areas of NYC where people frequently or rarely ride to, in at least 40 or 90 minutes? What is the average duration of such trips and how does the time of the year influence the rental behavior?*

The exemplified GrALa workflow for this analysis is shown in Listing 3.1. The input is the fully evolved CitiBike network as a single logical graph. First, we extract a snapshot containing only bike rentals happened in 2018 and 2019 with operator *Snapshot* (line 2-3). In lines 4-5, we add a specific `cellId` calculated from the geographical coordinates in its properties to each vertex (rental station) with the *Transformation* operator. The *Temporal Pattern Matching* operator in lines 6-14 uses the enriched snapshot to match all pairs of rentals with a duration of at least 40 or 90 minutes, each where the first trip starts in a specific cell (2883) in NYC and the second trip starts in the destination of the first trip after the end of the first trip. Since the result of the *Temporal Pattern Matching* operator is a graph collection we use a *Reduce* operator (line 15) to combine the results to a single

Name	SF	$\|V\|$	$\|E\|$	Size on disk
LDBC	1	3.3 M	17.9 M	4.2 GB
LDBC	10	30,4 M	180.4 M	42.3 GB
LDBC	100	282.6 M	1.77 B	421.9 GB
CitiBike	-	1174	97.5 M	22.6 GB

Table 3.5: Characteristics of the datasets used for the evaluation.

logical graph. We further group the combined graph by the vertex properties `name` and `cellId` (line 17) and the edge creation per month (line 19) using the *Time-dependent Grouping* operator. By applying two aggregation functions on the edges (lines 20-22), we determine both the number of edges represented by a super edge and the average duration of the found rentals. Finally, we apply a *Subgraph* operator to output only super edges with a count greater than 1 (line 23). The corresponding source code is public available at `https://git.io/JULPM`.

Figures 3.10, 3.11 and 3.12 show the performance results for both the execution of individual operators (Snapshot, Difference, Grouping and Pattern Matching (Query) for the LDBC dataset and for the analytical program on the real-world **C**iti **B**ike dataset (CBA40 and CBA90). We run each experiment five times and report average execution times. Figure 3.10 shows the impact of the LDBC data size on the runtime of individual operators. Figure 3.11 and Figure 3.12 show runtimes and speedup for different cluster sizes with LDBC SF 100 (for individual operators) and the CitiBike dataset (for the CBA program).

Snapshot and Difference Both temporal operators scale well for both increasing data volume (Fig. 3.10) and cluster size (Figs. 3.11 and 3.12). In general, the execution of *Difference* is slower than *Snapshot* since it has to evaluate two predicates as intermediate results and add a property to each element.

Figure 3.12 shows the speedup for LDBC.100 grows nearly linearly until 4 parallel workers and is slightly declining then for more workers. This behaviour is typical for distributed dataflow engines since a higher worker count increases the communication overhead over the network (especially for the semi-join performed in the verification

```
1  outGraph = citiBikeGraph
2    .snapshot(Overlaps('2017-01-01', '2019-01-01'))
3    .transform(v => v['cellId'] = getGridCellId(v))
4    .query("MATCH (v1:Station)-[t1:Trip]->(v2:Station)
5            (v2)-[t2:Trip]->(v3:Station)
6           WHERE v1.cellId == 2883 AND
7            v2.id != v1.id AND
8            v2.id != v3.id AND
9            t1.val.precedes(t2.val) AND
10           t1.val.lengthAtLeast(Minutes(X)) AND
11           t2.val.lengthAtLeast(Minutes(X))")
12   .reduce(g, h => g.combine(h))
13   .groupBy(
14    [label(),prop('name'),prop('cellId')],
15    (),
16    [label(),timestamp(val-from, MONTH)],
17    (superEdge, edges =>
18      superEdge['count'] = edges.count(),
19      superEdge['avgDur'] = edges.avgDur()))
20   .subgraph(e => e['count'] > 1)
```

Listing 3.1: CitiBike Analytical (CBA) program. (X defines the minimum duration of the bike rentals)

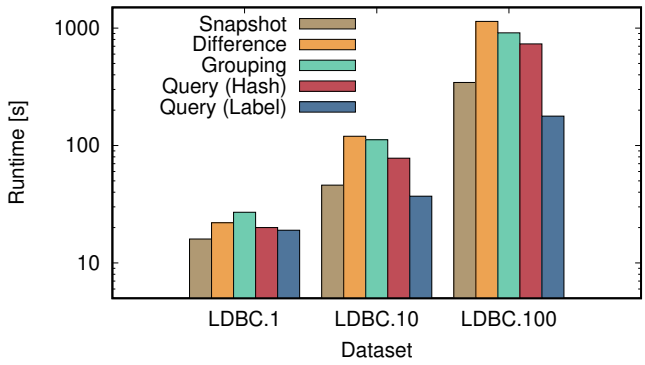


Figure 3.10: Average runtime of pipeline execution for five operators by increasing the data volume of the LDBC dataset.

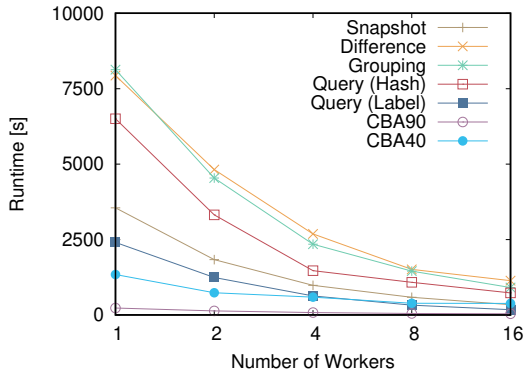


Figure 3.11: Average runtime of pipeline execution by increasing the cluster size, executed on the LDBC SF 100 dataset.

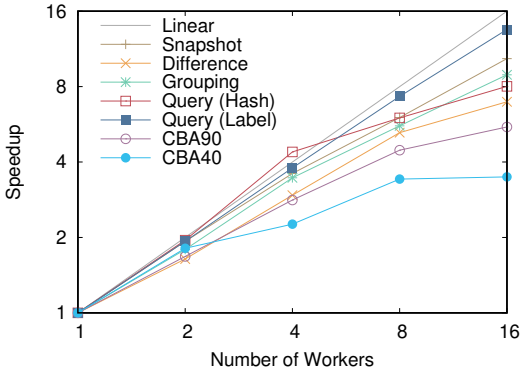


Figure 3.12: Speedup by increasing the cluster size, executed on the LDBC SF 100 dataset.

step) while the useful work per worker becomes smaller with larger configurations.

Grouping The evaluated *Grouping* operator on the LDBC datasets uses two grouping key functions, one on label values and one that extracts the week timestamp of the lower bound of the valid time interval. The grouping result thus consists of super vertices and edges that show the weekly evolution of created entities and their respective relationships over several years. For aggregation, we count the number of new entities and relationships per week.

Figure 3.10 shows that *Grouping* also scales nearly linearly for increasing data size (from LDBC.10 to LDBC.100) similar to the two previously discussed operators. The runtime reductions for smaller graphs are limited due to job initialization times. The execution time for LDBC.100 is reduced from 8,097 seconds on a single worker to 967 seconds on 16 workers (Figure 3.11) resulting in a speedup of more than 8. Figure 3.12 shows that the speedup for *Grouping* is almost linear for up to 8 workers while more workers result only in modest improvements. These results are similar to the original evaluation of the non-temporal *Grouping* operator in [111].

The shown performance results are influenced by the communication overhead to exchange data, in particular for the *join* and *group-by* transformations in our operator

implementations. This overhead increases and becomes dominant for more workers. In addition, the usage of a *ReduceGroup* transformation on the grouped vertices (see Section 3.4.3) can lead to an unbalanced workload for skewed group sizes. We already addressed this issue in [111], however, a more comprehensive evaluation of the influence of skewed graphs is part of future work.

Pattern Matching (Query) In this experiment, we execute a pre-defined temporal graph query on the LDBC data on various cluster sizes and two partitioning methods: First, hash-partitioned, which uses Flink's default partitioning approach combined with Gradoop's basic *GVE Layout* and second, label-partitioned, a custom partitioning approach which combines benefits of data locality and Gradoop's experimental *Indexed GVE Layout* (see Section 3.4.2). The used TemporalGDL query is shown below and determines all persons that like a comment to a post within partly overlapping periods.

```
MATCH (p:person)-[l:likes]->(c:comment),
      (c)-[r:replyOf]->(po:post)
WHERE l.val_from.after(Timestamp(2012-06-01)) AND
      l.val_from.before(Timestamp(2012-06-02)) AND
      c.val_from.after(Timestamp(2012-05-30)) AND
      c.val_from.before(Timestamp(2012-06-02)) AND
      po.val_from.after(Timestamp(2012-05-30)) AND
      po.val_from.before(Timestamp(2012-06-02))
```

Regarding the default hash partitioning, Fig. 3.10 shows that the execution of this query scales linearly with a larger data size from LDBC.10 (78 seconds) to LDBC.100 (733 seconds). Increasing the cluster size for LDBC.100 reduces the query runtime from 6,857 seconds for one worker to 733 seconds using 16 workers (Fig. 3.11). The speedup behavior (Fig. 3.12) is perfect for up to 4 workers and levels out for more workers to 7.8 for 16 workers due to increasing communication overhead resulting from semi-joins.

Applying the experimental label-partitioned approach, both the runtimes and the speedup results improve significantly compared to the use of hash partitioning. As shown in Fig. 3.10, the runtime improvements increase with growing data volume from a factor of 2 for LDBC.10 (37 instead of 78 seconds) to a factor of 4 for LDBC.100 (178 vs. 733 seconds).

This significant improvement is mainly achieved by a reduced complexity during semi-join execution, since our Indexed GVE Layout provides an indexed access via type labels. Therefore, Flink is able to optimize the execution pipeline by avoiding complex data shuffling and minimizing intermediate result sets. This is also confirmed by analyzing the impact of a growing number of workers on runtimes (Fig. 3.11) and speedup (Fig. 3.12) for LDBC.100. For 16 workers, our label-partitioned approach achieves an excellent speedup of 13.5 compared to only 7.8 with the hash-partitioned GVE Layout.

Overall, the results of the hash-partitioned experiment are similar to the ones of the other operators while the label-partitioned experiment significantly improves runtimes and speedup by reducing the semi-join complexity and communication effort. A more comprehensive evaluation including selectivity evaluation for different queries and partitioning approaches is left for future work.

City Bike Analysis (CBA) The results for the more complex analytical pipelines CBA40 and CBA90 are shown in Figure 3.11 and 3.12. Note that both analytical programs contain the same operator pipeline but with different rental durations for the query operator resulting in largely different result sets of 10M (CBA40) and 150K (CBA90) matches (i.e., matching subgraphs of the query graph) representing trips with a duration of at least 40 or 90 minutes.

The more selective CBA90 program could be executed in 223 seconds on a single worker and only 42 seconds on 16 workers, while CBA40 takes 1336 and 352 seconds for 1 and 16 workers, respectively. The sequence of multiple operators within the program leads to smaller speedup value compared to the single operators. This is especially pronounced for CBA40 leading to large intermediate results to be processed by the operators coming after the pattern matching operator. Intermediate results are graphs kept in memory as input for subsequent operators and the execution time of these operators is strongly dependent on the size and also the distribution of their input.

Generally, the size and distribution of the graph data has a significant impact on the analysis performance in Gradoop. Assume, we ask for a grouping of vertices on type labels but there are only 4 different type labels for vertices available. An execution of grouping on a cluster of 16 machines will see that only 4 machines of the cluster can be well utilized while the other machines remain largely idle. Such bottleneck operators can easily occur with analytical programs and limit the overall runtime. Resolving such problems would ask for more scalable implementations of operators, e.g. for a parallel grouping of one label on several workers, and for an optimized dynamic data distribution for intermediate graph results within analytical programs.

3.6 Conclusion

In this section we provided a system overview of Gradoop, an open-source graph dataflow system for scalable, distributed analytics of static and temporal property graphs. The core of Gradoop is the *TPGM*, a property graph model with bitemporal versioning and graph collection support, which is formally defined in this work. GrALa, a declarative analytical language enables data analysts to build complex analytical programs by connecting a broad set of composable graph operators, e.g., time-dependent graph grouping or temporal pattern matching with *TemporalGDL* as a query language. Analytical programs are executed on a distributed cluster to process large (temporal) graphs with billions of vertices and edges. Several implementation details show how the parts of the framework are realized using Apache Flink as distributed dataflow system. Our experimental analyses on real-world and synthetic graphs demonstrate the horizontal scalability of Gradoop. By applying a suitable custom partitioning, we where able to speedup the performance of our pattern matching operator by a large margin. Besides more performance optimizations and graph partitioning, future research directions we consider are: (i) using alternative technologies for Gradoop, its model and operators, (ii) extend our analysis from temporal graphs to graph streams, (iii) the integration of analysis using graph ML. We refer to Chapter 8 for a reflection on several lessons learned during our experience working on that project.

4 Gradoop Application Examples

The temporal analysis of evolving graphs is an essential requirement in many domains. With the previous introduction of the TPGM and the reference implementation Gradoop, a domain-independent possibility of modeling and analyzing such graphs is provided. This section presents two application examples of the TPGM and Gradoop. The modularity and combinability of analytical TPGM operators are demonstrated on a call center network in Section 4.1. Further, in Section 4.2, a web-based user interface called *Temporal Graph Explorer* is demonstrated using rental data of a bike-sharing provider as a running example. The contributions of both sections addressing the challenges C4, C7, C5, and C9 from Section 1.1.

4.1 Analyzing Call Center Data with Gradoop

This section provides a summary of the temporal graph grouping features implemented in Gradoop. This background information sets a base for the upcoming use case in the customer service domain. The use case serves to showcase the flexibility of the temporal graph model and its operators.

The contents of this chapter were already published under the title *Evolution Analysis of Large Graphs with Gradoop* [177].

4.1.1 Introduction

Temporal graphs represent the evolution of entities and relationships among them throughout time. Many real-world scenarios dynamically change over time, e.g., friendships and likes in social networks, citations and authorship affiliations in literature, or transactions between accounts in the financial domain [89]. Instead of neglecting this prevailing time dimension by using a static graph model, it is advisable to represent the continuously changing network in a temporal graph data model to enable studying the effect of time on the graph [215].

To provide an extensive framework for temporal graph analysis, we developed the previously introduced TPGM that enables modeling a graph with bitemporal time semantics and a set of operators to build distributed analysis workflows considering the additional time dimensions in the graph. We refer to Chapter 3 for further details of the model

and its implementation Gradoop. In the following, we first recap the time-dependent grouping operator, which is important for the following use case. We then show the expressiveness of the analytical language of Gradoop by composing new and existing operators to answer an analytical question from a call center use case.

4.1.2 Recap: Gradoop's temporal grouping

The temporal extension of Gradoop's *grouping* operator offers a flexible mechanism to group (summarize) vertices and edges, which belong to a given time instance. Users can either define their own or use predefined functions to extract keys from a vertex or edge on which to group. Any information of a graph element can be used including all temporal information, such as the day of the week on which the validity of an edge begins or the rounded duration of a vertex validity. Additionally, multiple aggregate functions can be specified to compute aggregates within a vertex or edge group and store them as a new property on the super-vertex (the vertex representing the group) or super-edge respectively.

Not only properties can be aggregated, but also information from the additional time dimensions of the graph. For example, the earliest or latest beginning of an edge validity or the average, minimum, or maximum vertex duration can be calculated. The resulting grouped graph is again temporal, i.e., the valid times of the super-vertices and -edges are defined by the earliest beginning and latest ending of the elements that are responsible to the group.

Since timestamp values can be analyzed and grouped at different granularities (e.g. year, month, day, hour, minute etc.), time properties inherently lead to hierarchically organized dimensions. Graph summaries determined by the *grouping* operator can thus be additionally "rolled up" on the time hierarchy to have aggregations on multiple levels of time granularity. A detailed description of graph grouping with Gradoop including the roll-up feature and predefined aggregate functions can be found in our GitHub wiki [172].

4.1.3 The call center use case

Supporting graph analysis at a large scale is necessary in various domains like Internet-of-Things (IoT), finance, and web to perform risk analysis, customer profiling, etc. In addition, time plays an important role in such analysis since analysts want to know, e.g., how a specific result of their query looks in the past or changes over time. As a result, a graph processing system has to offer a flexible and rich library of functionalities and algorithms to support a wide range of analyses respecting the additional time dimension.

To show the expressiveness and flexibility of Gradoop and its temporal model among its declarative operator principle, we choose a business case from the customer relationship management domain. Specifically, the scenario deals with interactions in a call center for 25 banks of the Banks Association of Turkey [203]. More than 7,500 agents are employed in about 16 service types (e.g., card, stock, ATM, online banking, etc.). Per month, about 46 million incoming calls are answered by agents, and 24 million calls are outgoing calls to customers. These entities and their relations form a huge heterogeneous network that continuously evolves.

Figure 4.1 shows a simplified example of the resulting graph schema. It includes different types of vertices (entities), like *Bank* and *Customer*, as well as edges (relations),

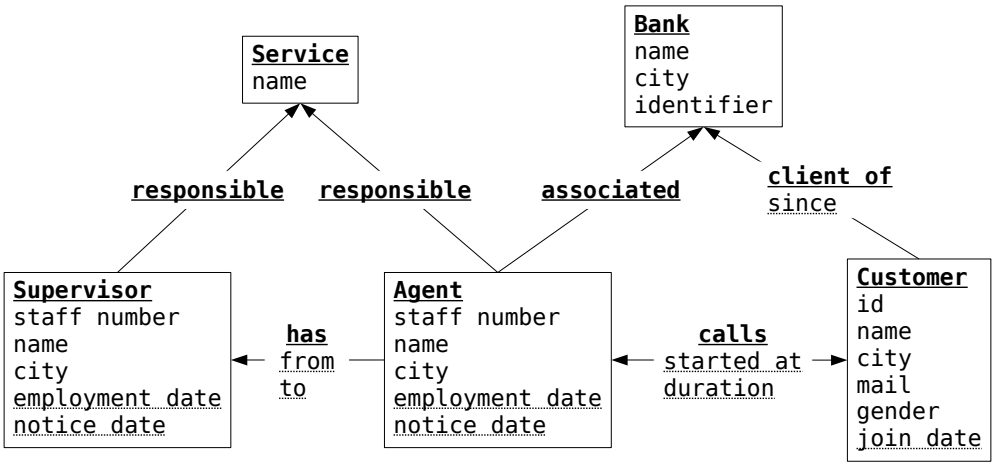


Figure 4.1: Simplified example of a call center network from the financial domain with underlined temporal properties.

like a *call* representing the telephone call between customers and call center agents. Each element includes a variety of properties describing it with additional information, e.g., an *Agent* vertex has a defined *staff number*, a *name*, and *city*. We can put all the collected data in our temporal property graph model. Properties containing temporal information (e.g., the *started at* and *duration* properties of the *calls* edge) can be directly mapped to the valid-time attributes of the model, to enable various time-related analyses.

In the following, we delve into the processing of an analytical question within this use case. Leveraging the modularity of our temporal graph operators, along with operators from the reference EPGM implementation, we will construct an analytical workflow. This demonstration aims to illustrate an effective method for addressing the analytical question at hand. Let's consider the following question: "*What is the average duration of calls per month, week and day between agents of different cities and customers of Istanbul, where both agents and customers joined the bank in 2018?*"

The question includes the need for aggregations over time hierarchies besides filters for a subset of entities on an extracted graph snapshot. The following exemplary workflow definition shows the use of four operators that result in a collection of graphs where each describes one out of the three time granularities month, week, and day.

```
groupedGraphs = graph
  .subgraph(
     v -> { v._label = 'Agent' OR
           (v._label = 'Customer' AND v.city = 'Istanbul' }),
     e -> { e._label = 'calls'})
  .snapshot(CreatedIn(2018))
  .verify()
  .groupBy(
     [Label(), Property('city')],                    // V grouping key func.
     [Count()],                                      // V aggregate func.
     [Month(from)), Week(from), Day(from)] BY ROLLUP, // E grouping key func.
     [AvgDuration(), Count()]                        // E aggregate func
  );
```

The initial *subgraph* operator (line 2 to 5) applies a filter using the given vertex and edge predicates to get a subgraph that contains only *Agent* vertices and *Customer* vertices with a property *city* that is equal to the string *Istanbul*. This operator is part of the EPGM. To receive customers who joined a bank in 2018, we apply the newly developed TPGM *snapshot* operator (line 6) with a predefined predicate. Since the result of the *snapshot* operator can contain dangling edges (i.e., their source or target vertices are not contained in the result set), we apply the *verify* operator (line 7) to remove these from the graph. The final *grouping* operator (line 8 to 12) summarizes the graph. The vertices will be grouped by their label and the property *city* (line 9). A property with the count is added to each grouped vertex as a result of the given *Count()* vertex aggregate function (line 10). The edges representing the calls are grouped by month, week, and day of the calls beginning timestamp (from) through the usage of time-specific value transformation functions of the same name (line 11). Since we want to know the average call duration, the predefined aggregate function *AvgDuration()* is specified in addition to the *Count()* aggregate function (line 12). Equivalent to the vertices, new properties storing the aggregates are added to each super-edge.

The additional `BY ROLLUP` (line 11) leads to three different aggregations comparable to SQL. First, the graph will be grouped by day, then by week, and, besides, by the month of the call's beginning. This leads to deeper insights into the evolution of the number and average duration of calls between agents of different cities and customers from the city of Istanbul. The resulting three graphs are contained in a graph collection, which is the result of our workflow and exemplified in Figure 4.2. Within the figure, each multi-edge graph represents one temporal granularity. For example, the edges of the lower graph are grouped by the month of their beginning timestamp. For simplicity, each grouped graph contains only a tiny subset of agents and call edges without temporal data. In the lower graph, 24 edges (twelve for each direction) exist between the grouped customer vertex

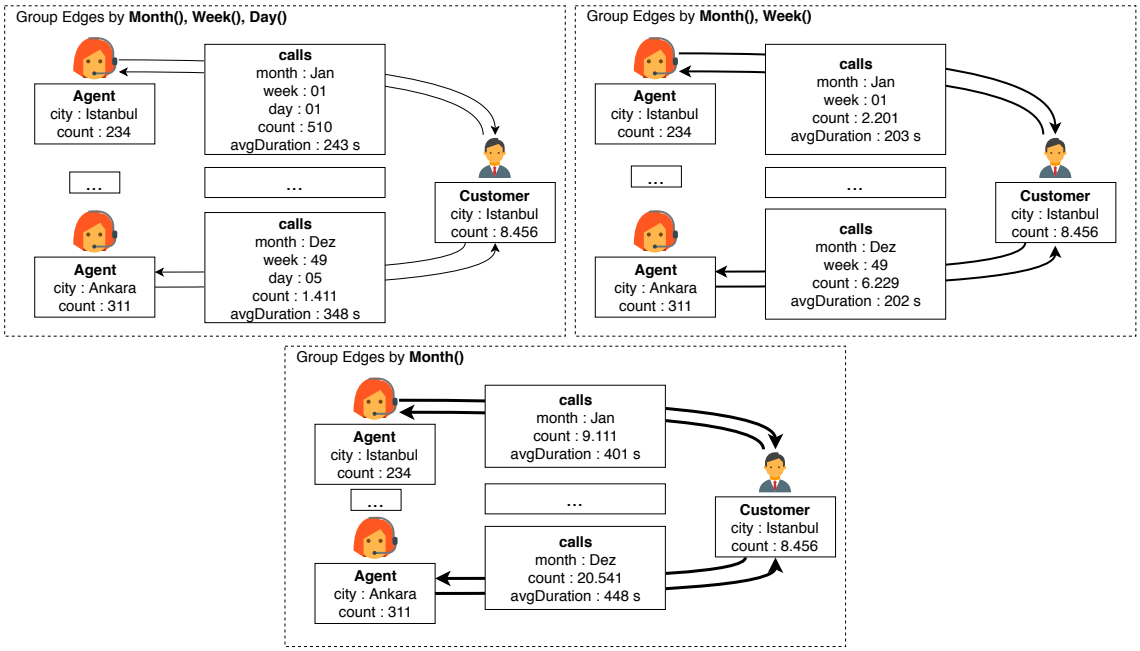
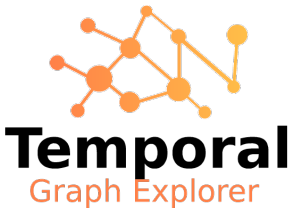


Figure 4.2: The resulting temporal graph collection from the graph analytical workflow.

and agents of a particular city. One of Gradoop's data sinks can store or visualize the collection. Further, an analyst may again use the *subgraph* operator to filter this result for periods with very low or high average call duration.

4.1.4 Conclusion

In this subsection, we showed how the temporal operators and further extensions of Gradoop enable a flexible answering of time-oriented analytical questions on temporal graphs by chaining several operators. We demonstrated the use of Gradoop's declarative workflows for a time-related use case scenario of the customer services domain. The described extensions are already implemented and available in Gradoop, which is comprehensively described in Chapter 3.



4.2 The Temporal Graph Explorer

In this section we introduce the *Temporal Graph Explorer (TGE)*, a distributed open-source framework that enables time-dependent graph exploration and analysis on large real-world networks using the TPGM and it's temporal graph operators. Besides retrieving a snapshot from a past graph state or calculating the difference between two graph snapshots, users can use the TGE to summarize the graph to reduce its complexity and to obtain deeper insights into its evolution.

The contents of this section were published under the title *Exploration and Analysis of Temporal Property Graphs* [175].

4.2.1 Introduction

Graphs are an intuitive way to model and analyze complex relationships between entities representing real-world scenarios. Since most entities and interconnections evolve in the real-world, graphs also change over time in terms of their structure and content. For example, Figure 4.3 shows a toy example of bicycle rentals (represented as directed edges) between fixed stations (vertices) over time. Such temporal property graphs [4] additionally allow tracking changes in the graph over time. In the example of Figure 4.3 both vertices and edges store temporal information (marked by a clock symbol) as attributes (valid times) and, thus, the graph reveals that the bike with id `2115` was moved from station `[1]` to `[2]` by three consecutive rentals over time.

In this paper, we demonstrate the *Temporal Graph Explorer (TGE)*, a tool for user-friendly exploration, analysis, and visualization of large temporal property graphs. The core of this application is Gradoop (see Chapter 3), an open-source framework for distributed graph analysis. Its *Temporal Property Graph Model (TPGM)* [178] enables modeling and analysis of graphs with bitemporal time semantics and comes with a set of composable temporal graph operators that can be combined with the help of the declarative language GrALa.

The TGE thus enables the analysis of the evolution of graphs, i.e., to figure out *when* something happened or changed, rather than a static view representing something that happened at some time [215]. To this end, it provides the execution of three temporal graph operators, namely *snapshot retrieval* and *graph difference*) to compute and visualize changes, including additions and deletions, that have been occurred. This can be used in

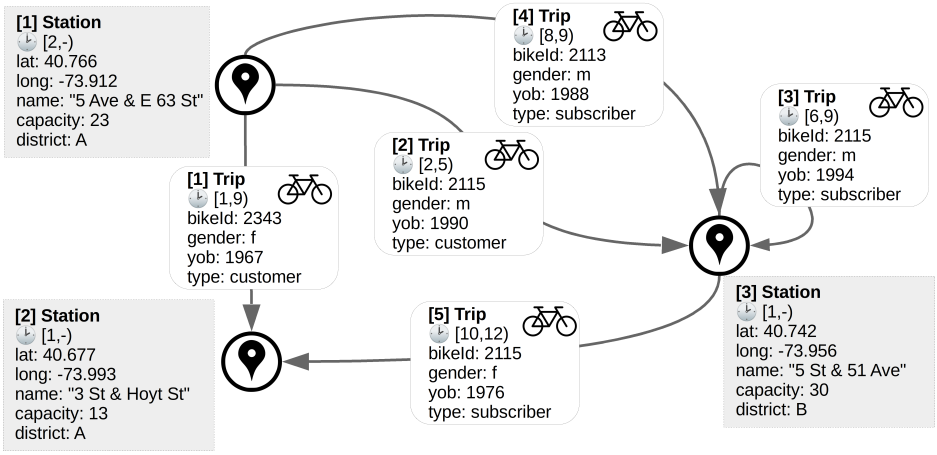


Figure 4.3: Example temporal property graph representing bicycle rentals between rental stations. The validity period of an edge is marked with a clock symbol and simplified with numbers instead of timestamps.

the given bicycle example to find, for a given week in the past, all rentals that have been added, removed, or remained the same.

The TGE can also be employed for the analysis of large graphs by using **time-related grouping and aggregation**. This allows for a profound exploration of a graph's structure, semantics, and development over time, which is a significant part of knowledge discovery for temporal graphs. Such a graph grouping mechanism helps to find out *how* different types of vertices and edges are connected as well as *when* and *how long* they were connected. In addition, the graph can be grouped on different dimensions, e.g., by rental time or by station location, as well as on different dimension levels, e.g., per year or month for the time dimension. The grouped vertices and edges can further be aggregated in any conceivable way, from a simple count to the minimum, maximum and average duration of a specific relationship type.

Section 4.2.2 gives a short view on related work, while Section 4.2.3 provides an system overview. Section 4.2.4 and Section 4.2.5 explain the usage of the operators snapshot, difference and time-related grouping and aggregation inside the TGE. Section 4.2.6 concludes this section.

4.2.2 Related Work

There are also other research that deals with the exploration of temporal graphs. Orlando et al. present the TGV [153], a framework for temporal property graph visualization that allows editing and running T-GQL [52] queries (see Section 2.4), displaying the result, and navigating such result across time. The focus is thus the temporal querying and result navigation. Compared to this, the TGE shown here focuses on extracting a snapshot from the temporal graph, generating a difference graph between two snapshots, and dynamically grouping the graph.

Aghasadeghi et al. [5] focus in their work on the temporal grouping. They propose two types of "zooming" (i.e., summarization/grouping), which is *attribute-based zoom* and *temporal window-based zoom* to support exploratory analysis of an evolving graph at dif-

ferent levels of resolution. It uses the TGraph data model of Moffitt and Stoyanovich [139]. The attribute-based zoom allows to model new temporal vertices from property values of existing vertices. The new vertices get time intervals for their validity and aggregated values like the count of the summarized vertices. This operation is similar to Gradoop's time-dependent graph grouping (see Section 3.3), where grouping key functions are used. The temporal window-based zoom allows the specification of time windows. The resulting graph reveals the information, which elements were in which window. The TGE provides similar functionality through grouping key functions that extract temporal features (e.g., the day of the week) from the time interval of vertices and edges to group them.

4.2.3 TGE System Overview

The *Temporal Graph Explorer* is an application to explore, analyze and visualize temporal property graphs. An intuitive web-based user interface enables the configuration of selected temporal graph operators and their application on a predefined graph dataset of the user's choice. The whole processing is then executed using Gradoop as a backend framework. The resulting graph is again temporal and sent back to the user interface for visualization. Frontend and backend communicate through a RESTful webservice.

The visualized graph is interactive, i.e., visitors can zoom in and out, drag vertices and edges to other positions or click on them to show their properties and temporal attributes. The visual representation is adjusted to the temporal characteristics of the graph and realized using Apache ECharts [68] and Cytoscape.js [72], an open-source software platform for visualizing complex attributed networks. By default, the coloration of vertices and edges is based on the respective label, i.e., elements with the same label are equally colored. Further, identifiers, properties, and temporal attributes are displayed in a tooltip after selecting a vertex or an edge. An architectural overview of the system is given in Figure 4.4.

The heart of the TGE is Gradoop, which offers many generic operators on graphs that can be used within workflows for graph analysis. To maintain the full history of a graph, including any insertion, deletion, or update of a vertex, edge, or its properties, Gradoop was extended by the recently introduced TPGM (see Section 3.3) as a graph data model. It supports labeled, directed multi-graphs where the vertices and edges are characterized by a unique identifier, a type label, and a set of properties, modeled as key-value pairs. In addition, each vertex and edge is extended by two time intervals τ^{val} and τ^{tx} that define a lifespan regarding to valid- and transaction time dimension, which ensures a bitemporal data modeling [105]. All further details about Gradoop can be found in Chapter 3.

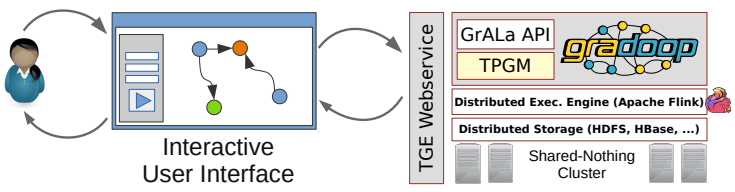


Figure 4.4: System architecture overview of Temporal Graph Explorer.

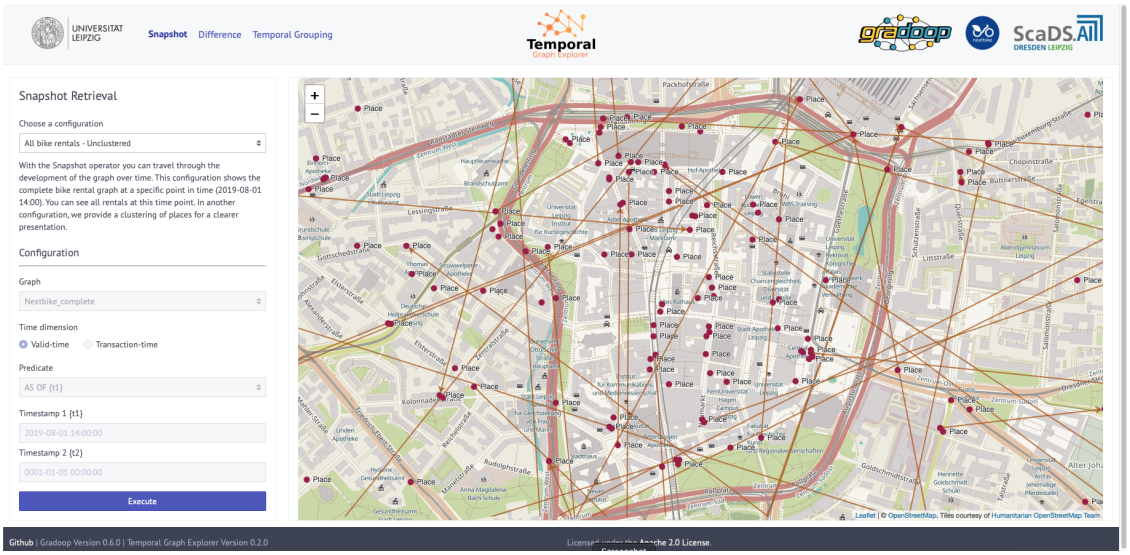


Figure 4.5: Screenshot of the TGE showing the snapshot view.

4.2.4 Graph snapshot and difference

To enable temporal and evolutionary queries and analysis, one data management challenge for large historical graphs is the retrieval of graph snapshots as of any time-point in the considered time domain [113]. To achieve this, we developed the *snapshot* operator that can be applied on a temporal graph instance and allows to retrieve a valid state of the graph either at a specific point in time or a subgraph that is valid during a time range. The user can configure the operator by pre-defined time-dependent predicates such as *asOf()*, *overlaps()* or *precedes()*. Furthermore, user-defined predicates, as well as helper functions to extract certain time dimensions, can be used. Through the visualization, the user receives instant feedback on the changes made and thus can explore and compare various states of different times.

Figure 4.5 shows a screenshot of the TGE with selected snapshot operator. On the left menu, the user chooses the graph dataset, the time dimension (here *Valid-time*) and the predicates to configure. After pressing *Execute*, the snapshot graph is visualized. Since the vertex data contain real coordinates as properties, the visualization of the graph uses a geographical layout to place vertices on a map.

An important part of the analysis of graphs is the examination of changes that have occurred between two points in time. Changes, i.e. additions, deletions, and edits, represent the evolution of a temporal graph and can be selected or aggregated in subsequent analysis steps. Therefore, we demonstrate the *difference* operator that computes a graph $\triangle G$ between two graph snapshots G_1 and G_2 by determining the union $G_1 \cup G_2$ and extending each element by a property that expresses the addition, deletion, or persistence of this element respectively. The user configures both snapshots by using time-dependent predicate functions, as described before. In addition, the desired time-dimension can be selected. For the visualization, graph elements are colorized depending on the annotation with which the elements are expanded by the difference operator to provide more information about their temporal evolution, which can be seen in Figure 4.6. A vertex or edge is colored red if the elements have been deleted in the time between both snapshots, grey if the elements were not changed at all, or green if the elements were created. Using

this kind of visualization, a user gains insights about how and how frequently the graph evolves between two states.

4.2.5 Time-specific grouping and aggregation

The maintenance of the entire history of a real-world graph entails a huge amount of graph elements. For example, the real-world citibike dataset we used in Section 3.5 contains up to 97.5M edges. A structural grouping of the graph (also denoted as summarization) will help to reduce the overall complexity and offers deeper insights into the graph's structure, distribution, and evolution. For example, a graph with billions of vertices and edges can be first grouped by the element's label to explore the schema that reveals how the heterogeneous types are connected. In addition, temporal and content information of the grouped vertices and edges can be aggregated in many ways to get knowledge about the respective groups.

The temporal grouping operator of Gradoop goes further and offers a flexible mechanism to group a temporal property graph by *all* available information of the vertices and edges, especially their temporal characteristics. This is achieved by the possibility of defining a function $f(v) \mapsto k$, denoted as *key function*, that maps a vertex v (or edge) to a key k on which to group. All elements mapped to the same key are grouped together and form a new super-vertex or -edge, respectively. To simplify the specification for users, Gradoop offers predefined key functions, e.g. label() to group elements by their label, as well as helper functions, e.g., functions for extracting time-related information to summarize the graph at different temporal resolutions. Since real-world graphs are usually very heterogeneous, the application of the key functions can also be restricted to nodes or edges of a certain label (what we call *label-specific grouping*), e.g., to group *Station* vertices by district, *Person* vertices by gender and *Trip* edges by bike.

Besides the grouping itself, one main feature is to enrich the grouped elements by summarized information about the group, which can be achieved by applying pre- and user-defined aggregate functions. Not only properties but also information from the additional time dimensions can be aggregated. For example, the earliest or latest beginning of an edges validity can be calculated by using *minTime()* or *maxTime()* aggregate functions.

The configuration options of this operator are very extensive and depend on the characteristics of the selected graph dataset and the objectives of the corresponding

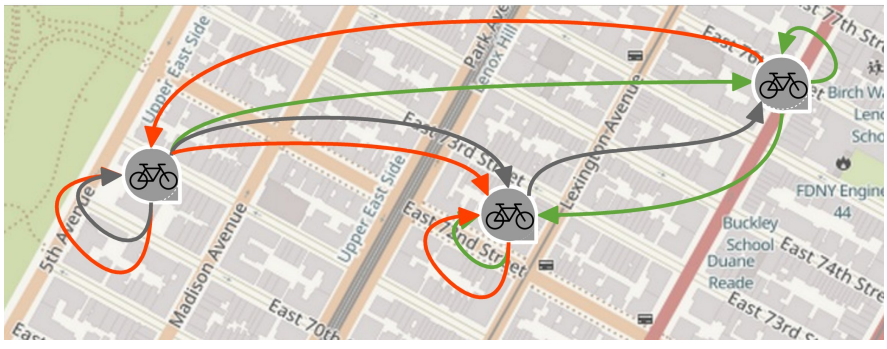


Figure 4.6: Example visualization of a difference graph. Elements, that are in the first snapshot, but not in the second, are colored red. Elements, that are in both snapshots, are colored grey. Elements that are just in the second snapshot are colored green.

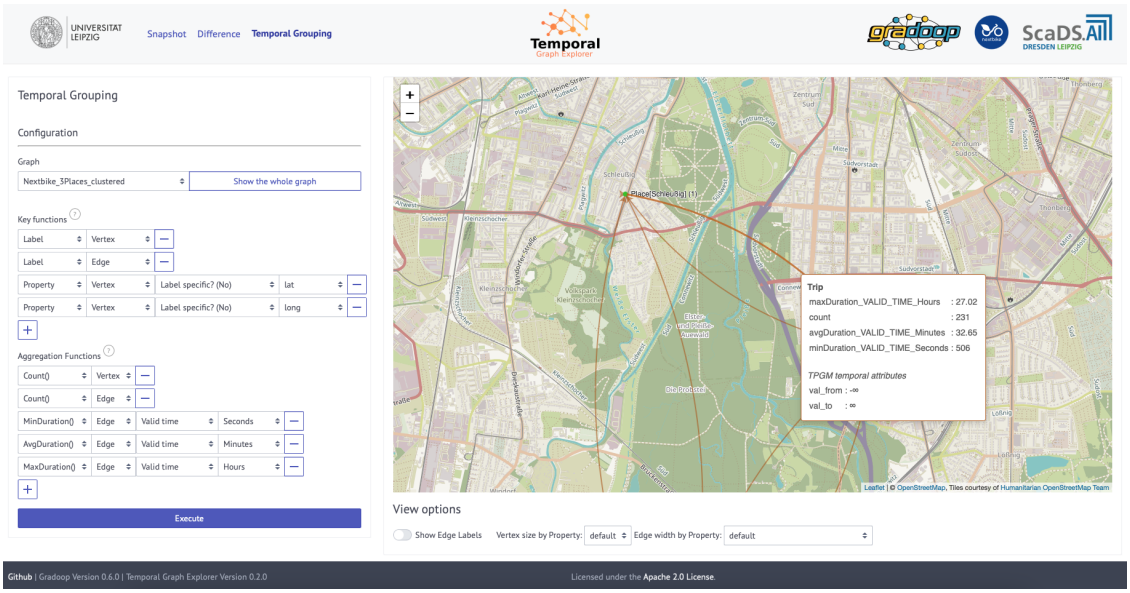


Figure 4.7: Screenshot of the TGE showing the grouping view.

analysis. The *Temporal Graph Explorer* supports the user in the configuration of the operator by a flexible selection of the predefined key functions for vertices and edges, as well as aggregate functions, which can be seen in the screenshot in Figure 4.7 on the left side. Appropriate arguments are offered for parameterized key functions. For example, a list of property names is offered for the function `property(<name>)`. Timestamps which appear to be useful for the selected temporal graph are also suggested for use with temporal key functions. Besides, the user can choose from pre-defined aggregate functions to additionally configure the grouping and thus to enrich the grouped elements with detailed information about the grouped element, which can be accessed in the graph visualization. The user can thus interactively add key and aggregate functions to the configuration step by step until the grouped graph and its aggregated values provide information about a specific analysis question. Since the grouped vertices have geographic properties (see that two key-functions are selected, that group vertices on the properties latitude (lat) and longitude (long)), they can be placed on a map-view. Edges can also be colored according to a certain property. The properties created by the aggregates are displayed after selecting a vertex or an edge, as can be seen in Figure 4.7.

For the visualized result of the grouping operator, aggregated properties (e.g., `count`, `minDuration`, `maxTimestamp`) can be used to adjust the radius of vertices and width of edges to the corresponding property value. This configutation possibility can be seen on the bottom of Figure 4.7. For example, the width of a super-edge could depend on the average duration of the grouped edges.

4.2.6 Conclusion

In conclusion, this section has introduced the Temporal Graph Explorer (TGE), a powerful tool designed for the exploration, analysis, and visualization of large temporal property graphs. By leveraging the Temporal Property Graph Model (TPGM), the TGE enables the study of evolving graphs. Rather than a static view, it allows us to delve into the dynamic

nature of graphs, uncovering when and how changes occur.

The TGE's functionality includes essential temporal graph operators such as snapshot retrieval and graph difference, facilitating the computation and visualization of changes, additions, and deletions within the graph. This is particularly valuable for scenarios like the bicycle rental example, where one can pinpoint rentals that have been added, removed, or remained unchanged during a specific time frame.

Furthermore, the TGE supports the analysis of large graphs through time-related grouping and aggregation, enhancing our understanding of a graph's structure, semantics, and development over time. This grouping mechanism enables the exploration of various aspects, such as how different types of vertices and edges are connected, when they were connected, and for how long. The tool's flexibility extends to grouping dimensions and levels, allowing users to aggregate data in ways that suit their specific analysis needs.

The Temporal Graph Explorer represents a significant contribution to the field of temporal graph analysis, offering a user-friendly and efficient means to gain insights into the evolution of complex, time-dependent relationships in various domains.

5 Evolution of Degree Metrics

Graph metrics, such as the simple but popular vertex degree and others based on it, are well defined for static graphs. However, adapting static metrics for temporal graphs is still part of current research. In this paper, we propose a set of temporal extensions of four degree-dependent metrics, as well as aggregations like minimum, maximum, and average degree of (i) a vertex over a time interval and (ii) a graph at a specific point in time. We show why using the static degree can lead to wrong assumptions about the relevance of a vertex in a temporal graph and highlight the need to include *time* as a dimension in the metric. We propose a baseline algorithm to calculate the degree evolution of all vertices in a temporal graph and show its implementation in a distributed in-memory dataflow system. Using real-world and synthetic datasets containing up to 462 million vertices and 1.7 billion edges, we show the scalability of our algorithm on a distributed cluster achieving a speedup of around 12 on 16 machines.

The contents of this section were published under the title *Evolution of Degree Metrics in Large Temporal Graphs* [174].

5.1 Introduction

Temporal graphs are graphs that change in structure and content over time, where changes are captured and maintained as part of the graph data model. Many approaches exist to formally define a temporal graph [83, 100, 117, 176]. A graph's evolution is either represented as a series of snapshots, or by vertex and edge annotations for timestamps or time intervals describing their validity. These extended graph models allow analyzing the current or a past state of a graph as well as the evolution of the graph. Examples for temporal graph analysis are the exploration of human contact networks to detect the transmission of a disease [168, 190] or analyzing the change in the utilization of bike rental stations [121, 206]. In such graphs, the concepts of graph metrics also change because time is added as a new dimension. Metrics used for the characterization of static graphs need to be redefined or extended to take temporal evolution into account [145].

One simple yet important metric of a vertex is the *vertex degree* [82]. It is determined by the number of incoming and outgoing edges (which is, except for multigraphs, equal to the number of neighbors) and thus a simple indicator for the relevance or importance of a

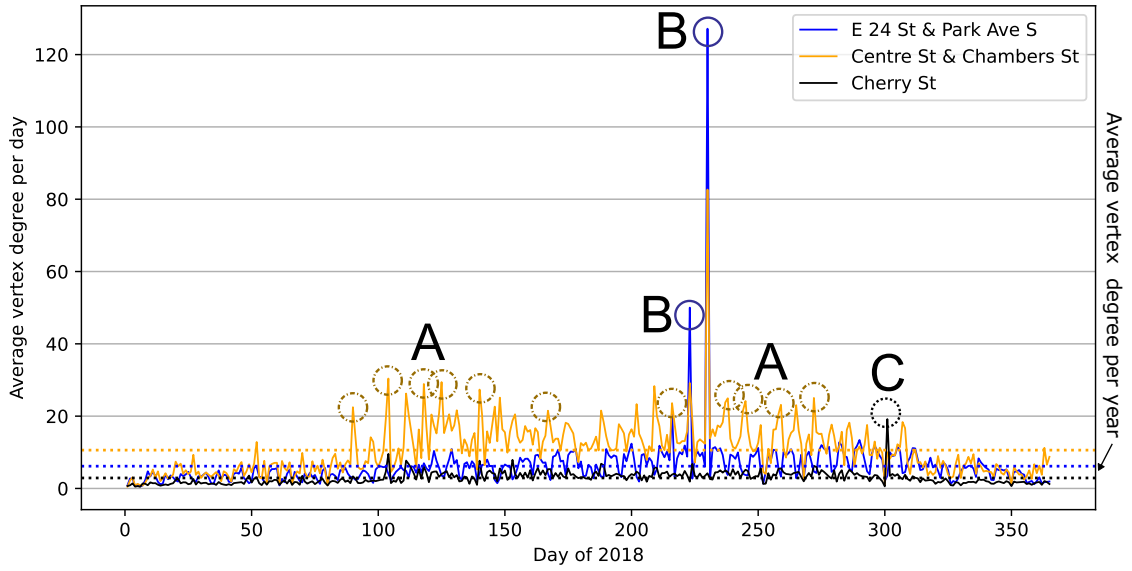


Figure 5.1: Degree evolution of selected rental stations in NYC for 2018. For each day, the average degree is plotted. **A** indicates peaks on weekends, **B** a construction embargo event and **C** a Halloween parade.

vertex in a static graph. A vertex with a high degree can be seen as a strongly connected vertex, whereas a vertex with a degree of zero is an isolated vertex or singleton. The vertex degree is also known as the centrality measure *degree centrality* [73], that can be used to find, for example, popular people according to their number of friendships in a social network, or the stations with the highest throughput of bike rentals in a bike-sharing network.

The minimum and maximum degrees are metrics that describe the vertices with the smallest and largest numbers of connections, respectively. The *degree range* [125], *degree variance* [125, 193, 194] and the *average nearest neighbor degree* (ANND) [94, 125], are aggregate metrics that can reveal important graph and vertex characteristics. The *degree range* of a graph (the difference between the maximum and minimum degree) describes the connectivity gap between the best and least connected vertices. For a bike-sharing network, a small degree range indicates a good distribution of rental stations without any hardly visited stations, whereas a high degree range indicates irregular usage. Another extended measure of a graph's heterogeneity is the *degree variance*, where a high variance shows a high inequality in the connectivity of the vertices. The *ANND*, on the other hand, reveals if a vertex is connected to others with a high connectivity, e.g., a social network user who is mainly friend with other users who are strongly connected.

Using only the static vertex degree is of limited value in an evolving graph as it cannot reflect the impact of topology changes. The same restriction applies for static aggregated metrics such as the average degree value [114] or the sum of all degrees [204]. There is no information about *when* a vertex has what degree, *how long* this degree is valid, and *when* it increases or decreases. This is important, for example, in a bike-sharing network where vertices represent stations and directed edges connect the start and return stations of bike rentals.

Figure 5.1 shows the time series representing the evolution of the vertex degree of

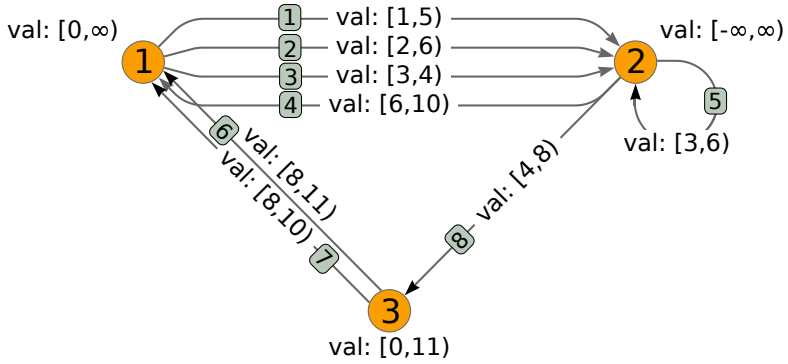


Figure 5.2: An example temporal graph.

three selected bike rental stations in NYC for 2018, calculated from the publicly available dataset also used in our evaluations (see Section 5.5). For example, one can see the popularity of the station at Centre St & Chambers St on weekends by periodic peaks (marked with **A**) or the significantly higher rental rate of two stations during the Summer Streets Construction Embargo [46] in August (marked with **B**). Further, the impact of a Halloween parade [195] on Cherry St. (marked by **C**) is visible in these time-series. This shows that there are stations that are generally popular, such as in a city center or near train stations, as well as stations that are only popular at certain times, e.g., on weekends or during events. Further, comparing stations using the static or aggregated metrics, which are shown in Figure 5.1 as dotted lines, may lead to the assumption that they seem equal by sharing a similar degree value, which in fact is not true over time which can be revealed by temporal metrics.

Figure 5.2 shows a toy example of a temporal graph, which we use to illustrate the problem further. Each vertex and directed edge has a unique numeric identifier and a left-close right-open time interval $[\omega_a, \omega_b)$[1] assigned. For example, the edge with identifier 5 (hereinafter referred to as e_5) is valid from time point 3 (in the following denoted as ω_3) to ω_6, whereas the vertex v_1 is valid from ω_0 to the maximum upper bound, denoted by the infinity symbol ∞ (ω_{max}).

From a static perspective, if we disregard the graph's evolution, we can see that the vertex degrees are $deg(v_1) = 6$, $deg(v_2) = 7$, and $deg(v_3) = 3$. However, if time is considered, then the degree values change continuously so that the evolution of the degree value forms a time series. For example, at time ω_1, the degree of v_1 is 1, and the same at time ω_5. Further, since v_1 is valid until forever and the last validity of its edges end at time ω_{11} (exclusive), the degree from ω_{11} to forever (ω_{max}) is 0. Figure 5.3 exemplifies the evolution of the degrees of v_1, v_2 and v_3, inclusive in- and outdegree of v_1 ($deg^-(v_1)$ and $deg^+(v_1)$).

It can be seen that the maximum degree of vertex v_1 is only 3 over its entire period of validity. From the vertex lower bound ω_0 to time ω_1, the degree is 0 – the same from ω_{11} onwards. Compared to the static point of view, where the degree is 6 for v_1, we can see that during the evolution of the graph the vertex never reaches this value. The same holds for the bike rental example of Figure 5.1. For example, the static degree of the station "Cherry St." is 50, whereas the maximum value over the year is just 20 for a single day. This shows the importance of considering the changes of the degree metric over

[1]For simplicity we use integer interval bounds. It holds $[\omega_a, \omega_b) := \{\omega \in \mathbb{N} : \omega_a \leq \omega < \omega_b\}$.

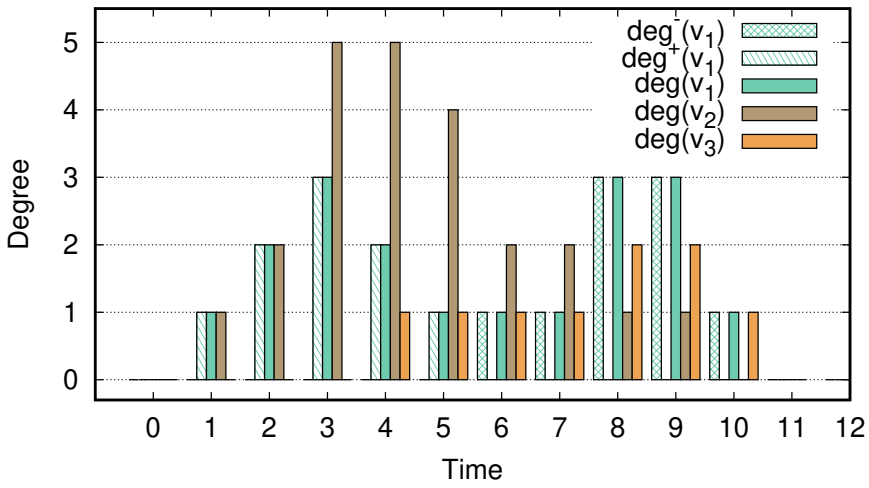


Figure 5.3: Degree evolution of vertex v_1, v_2 and v_3 from ω_0 to ω_{12}. In addition, the indegree $deg^-(v_1)$ and outdegree $deg^+(v_1)$ are given for v_1.

time. The use of the static degree metric for assessing the importance of a vertex can lead to misinterpretations, whereas using the degree evolution provides the exact degree for any time in the lifetime of the graph.

Contributions: In this work we focus on four time-sensitive degree-dependent graph measures: the vertex degree itself and its aggregations, the degree range, the degree variance, and the average nearest neighbor degree. We extend these well-known static metrics with a time dimension and establish two new formal definitions per metric: (i) a temporal version that defines the metric at a specific point in time, and (ii) an evolutionary version that defines the change of the metric within a time interval as a time series. We then present a baseline algorithm that can calculate the degree evolution for all vertices of a given temporal graph. Using a binary search tree called *degree tree*, the algorithm efficiently maintains the degree changes of each vertex. We show how our algorithm can be adapted to a distributed processing model, which is further illustrated by the implementation as a graph analysis operator using a distributed in-memory dataflow system. In our experiments, we evaluate the scalability of our implementation which shows a sublinear growth of runtime by increasing dataset size as well as a speedup of up to 12 on a cluster with 16 physical machines. The contributions address the challenges C4, C6, and C8, from Section 1.1.

Related work Some works have defined a degree metric for vertices in a temporal graph, mainly by expanding the static version for temporal graphs. Thompson et al. [204] introduce a *temporal degree centrality* metric for the domain of network neuroscience. They show that a node's influence in a temporal network can be represented by the centrality metric, which is the sum of the number of edges across a series of time points. If an edge is valid for multiple time points, it will be counted multiple times. However, this approach does not quantify the temporal order of edges so that different vertices with identical metrics cannot be distinguished.

A similar definition of temporal degree centrality is given by Long et al. [127] and Wu et al. [221]. Both calculate the sum of degrees over a time interval, which provides

an estimate of a node's centrality in a temporal network. Wang et al. [217] propose the *temporal degree deviation centrality* metric that can be calculated from a temporal network using graph snapshots. A similar approach defines the temporal degree as the number of nodes to which a vertex is linked in all timestamps of an interval without interruption [45].

The *time-ordered graph* model by Kim et al. [114] can represent a dynamic network with a fixed vertex set and interval edges. For graphs of this model, several centrality metrics were introduced (including degree) to include the graph's temporal characteristic. Temporal degree is defined as the degree $D_{i,j}(v)$ for a vertex $v \in V$ in a time interval $[i, j]$. Tlebaldinova et al. [206] use the degree as a temporal measure of centrality for bike-sharing stations. They show that the changing degree determines the time-distributed intensity of incoming and outgoing bike flows at a station.

In all these related works, the temporal degree is mostly seen as a scalar, aggregated (summed) value over a certain time interval, that is used as a centrality measure. In our approach, described next, we define both a *temporal degree* at a specific point in time as well as *degree evolution* for a time interval as a time series. This allows exact statements when a metric has what value for how long. In addition, our data model allows both changes in vertices and edges, as we describe in Section 5.2.1.

After the publication of this work, Andriamampianina et al. [12] introduced the *semantic temporal degree*, a centrality metric that integrates both temporal and semantics aspects. With semantic aspects, they mean the type label of the incident edges. A vertex degree thus does not just depend on the time, but also on the label of its edges. Besides the *average semantic temporal degree* they also defined the *semantic temporal degree distribution*.

5.2 Degree-dependent metric evolution

We first define the temporal graph data model we use as a basis for our work in Section 5.2.1, and then introduce new temporal notations of degree-dependent metrics for vertices in Section 5.2.2, and metrics for a whole temporal graph in Section 5.2.3.

5.2.1 Temporal graph model

We use a simplified version of the *Temporal Property Graph Model (TPGM)* data model, which was already introduced in Section 3.3. Although the model supports bitemporal versioning, for simplicity we limit ourselves to one time dimension. Thus, vertices and edges are assigned with a left-closed right-open time interval to represent the element's validity according to application-specific valid-time. Unlike most temporal graph models [39], not only the edge set is dynamic, but the vertex set can also change over time. Contact sequence graphs [90] can also be modeled by representing the time ω_i of the contact as time interval $[\omega_i, \infty)$, $[\omega_i, \omega_{i+1})$ or $[\omega_i, \omega_j)$ (depending on the use-case), where ω_j is the time of a subsequent contact. The formal definition of a TPGM graph, including consistency constraints, is given in Section 3.3.

As an addition to the model definition, we introduce two time-dependent sets of nodes and edges that we use later in the formal definitions in Section 5.2.2 and Section 5.2.3:

- $V(\omega_i) \subseteq V$ is a finite subset of vertices, where each vertex is valid at the given time point ω_i, i. e., for all $v = \langle v_{id}, \tau \rangle \in V(\omega_i)$ with $\tau = [\omega_{start}, \omega_{end})$ it holds: $\omega_{start} \leq \omega_i < \omega_{end}$.
- $E(\omega_i) \subseteq E$ is a finite subset of edges, where each edge is valid at the given time point ω_i, i. e., for all $e = \langle e_{id}, s_{id}, t_{id}, \tau \rangle \in E(\omega_i)$ with $\tau = [\omega_{start}, \omega_{end})$ it holds: $\omega_{start} \leq \omega_i < \omega_{end}$.
- $G(\omega) = (V(\omega), E(\omega))$ is a graph snapshot (or state) of a temporal graph G at a specific point in time ω.

5.2.2 Vertex-centric temporal degree metrics

For each of the following degree-based metrics, we first refer to the static version and then introduce our temporal and evolutionary version of the respective metric.

Vertex degree and aggregations. According to graph theory [56, 82], the static (non-temporal) **vertex degree** $deg(v)$ is formally defined as follows:

Definition 10 (Vertex degree [56, 82])**.** *The **degree** (or valence) of a vertex v in a static graph $G = (V, E)$, denoted $deg(v)$, is the number of proper edges incident to v plus twice the number of self-loops. Simplified, the degree of a vertex is the number of its edges. The **indegree** of a vertex v, denoted as $deg^-(v)$, is the number of edges directed to v whereas the **outdegree** of vertex v, denoted as $deg^+(v)$, is the number of edges directed from v. Each self-loop at v counts one toward the indegree of v and one toward the outdegree.*

Having a static view on the graph of Figure 5.2, example vertex degrees are $deg(v_1) = 6$, $deg(v_2) = 7$, $deg^+(v_2) = 3$, and $deg^-(v_3) = 1$.

For temporal graphs, we now define the *temporal degree* as the degree of a vertex *at a specific point in time.*

Definition 11 (Temporal degree)**.** *The **temporal degree** of a vertex v in a temporal graph $G = (V, E)$, denoted as $degt(v, \omega)$, is the degree of that vertex at time ω in the graph snapshot $G(\omega)$. It is defined as:*

$$degt(v, \omega) \begin{cases} deg(v), & \text{if } v \in V(\omega), \\ not\ defined, & \text{otherwise.} \end{cases} \tag{5.1}$$

*If $v \notin V(\omega)$, the degree is not defined. Analogous to the static degree, the **temporal indegree** $degt^-(v, \omega)$ is the number of edges directed to v, and **temporal outdegree** $degt^+(v, \omega)$ is the number of edges directed from v, at time ω.*

For example, in the graph of Figure 5.2, the temporal degree of vertex v_1 at time ω_4 is $degt(v_1, \omega_4) = 2$, whereas the temporal indegree of vertex v_1 at time ω_8 is $degt^-(v_1, \omega_8) = 3$. There are clear differences between the static compared to the temporal metrics.

From the perspective of a vertex v, the degree of that vertex changes according to the existence of neighbours of v. For a given time interval τ, we thus define the *degree evolution* as a time series of temporal degrees, which contains all degree values with their corresponding time in the given interval.

Definition 12 (Degree evolution)**.** *The **degree evolution** $degev(v, \tau) := \{x_1, x_2, ..., x_m\}$ of a vertex v is a time series of elements $x_i := degt(v, \omega)$, with $1 \leq i \leq m$ and $m = \omega_{end} - \omega_{start}$.*

*Each x_i represents a temporal degree at time ω_j, i.e., x_1 at time point ω_{start} and x_m at $\omega_{end}-1$, for the interval $\tau = [\omega_{start}, \omega_{end})$. Further, the temporal degree is a special case of the degree evolution: $degev(v,\tau) = \{degt(v,\omega_i)\}$ with $\tau = [\omega_i, \omega_{i+1})$ as an interval with a single time point. Furthermore, $degev^+(v,\tau)$ denotes the **outdegree evolution** whereas $degev^-(v,\tau)$ denotes the **indegree evolution**.*

For our example graph of Figure 5.2, the degree evolution of vertex v_1 in the interval $\tau = [\omega_0, \omega_{11})$ is $degev(v_1,\tau) = \{0,1,2,3,2,1,1,1,3,3,1\}$.

The degree evolution defines the development of a vertex degree over a given time interval. This can now be used to determine the minimum, maximum and average degree of *a vertex* over a time interval, i.e., a vertex-centric aggregation.

Definition 13 (Vertex-centric min/max/avg degree)**.** *The **vertex-centric minimum degree** of a vertex v within a time interval τ is the smallest value of all temporal degrees of v in this interval. Similarly, the **vertex-centric maximum degree** is the largest value and the **vertex-centric average degree** is the average value over all time points $\omega \in \tau$. With $|\tau|$ as the number of all time points in the interval τ holds:*

$$deg_{min}(v,\tau) := min\{degt(v,\omega)|\forall\omega \in \tau\}, \tag{5.2}$$

$$deg_{max}(v,\tau) := max\{degt(v,\omega)|\forall\omega \in \tau\}, \tag{5.3}$$

$$deg_{avg}(v,\tau) := \frac{1}{|\tau|}\sum_{\omega\in\tau} degt(v,\omega). \tag{5.4}$$

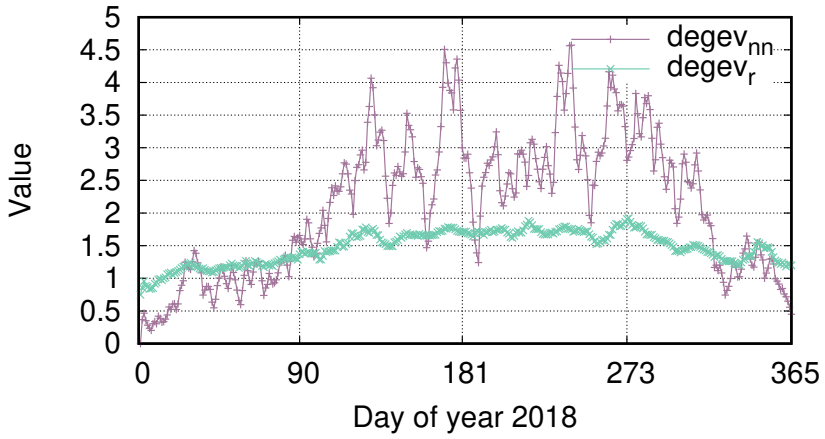
Average Nearest Neighbor Degree. An analyst may be interested in whether entities in a graph tend to connect to others with a high connectivity, or, the opposite case, connections occur randomly and irrespective of the degree [125]. The former situation is referred to as *preferential attachment* in network science [103] and applies to many real-world networks [35, 144], including evolving networks [103]. A metric to measure this tendency is the *average nearest neighbor degree* (ANND) $deg_{nn}(v)$. For a vertex v, the ANND is the sum of the direct neighbor degrees divided by the degree of v.

Definition 14 (Average nearest neighbor degree [125])**.** *The **average nearest neighbor degree** $deg_{nn}(v_i)$ of a vertex v_i of a static graph G is defined as the sum of the degrees of each of the vertex' neighbor v_j divided by the degree of v_i:*

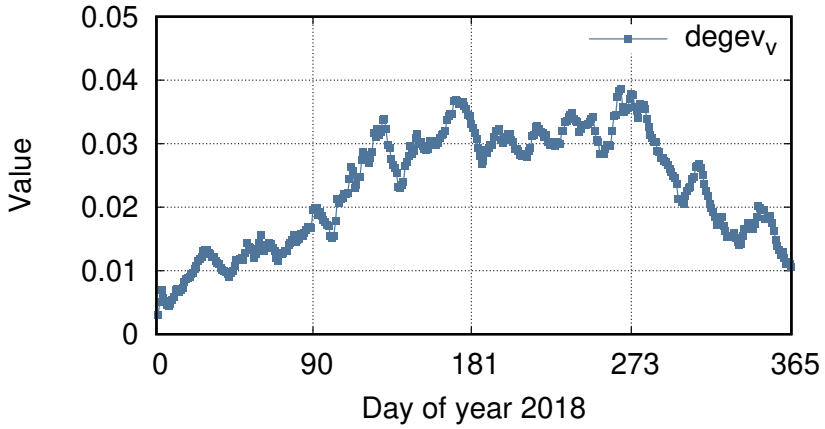
$$deg_{nn}(v_i) := \frac{1}{deg(v_i)}\sum_{v_j\in N(v_i)} deg(v_j). \tag{5.5}$$

The set $N(v_i) \subset V$ is defined as the set of vertices incident to a vertex v_i (its neighbors).

From a static perspective of the example graph of Figure 5.2, the ANNDs are $deg_{nn}(v_1) = \frac{deg(v_2)+deg(v_3)}{deg(v_1)} = 1.67$, $deg_{nn}(v_2) = 1.43$ and $deg_{nn}(v_3) = 4.34$. These results suggest that vertex v_3 seems to have the strongest tendencies to connect to others who are also popular, while v_1 and v_2 display weaker tendencies. The average degree of the graph (here $deg_{avg} = 5.34$) can be used to interpret an ANND value. The larger the value compared to the average degree of the graph, the more likely we can assume that its neighbors are more popular than average. As the other degree-dependent metrics, the ANND will change over time if a graph evolves. To calculate the ANND of a vertex at a specific point in time, we now define the *temporal average nearest neighbour degree*:



(a) ANNDE and degree range evolution.



(b) Degree variance evolution.

Figure 5.4: Resulting time-series of selected degree evolution metrics of dataset citibike for year 2018.

Definition 15 (Temporal ANND). *The **temporal average nearest neighbor degree** (TANND) $degt_{nn}(v_i, \omega)$ of a vertex v_i is defined as the sum of the temporal degrees of each of the vertex' neighbor (at time ω) divided by the temporal degree of v_i. Furthermore, the set $N(v_i, \omega) \subset V(\omega)$ is defined as the set of neighbors of vertex v_i at time ω. It then holds:*

$$degt_{nn}(v_i, \omega) := \frac{1}{degt(v_i, \omega)} \sum_{v_j \in N(v_i, \omega)} degt(v_j, \omega). \tag{5.6}$$

For the example in Figure 5.2, the TANND for v_1 at time ω_4 is $degt_{nn}(v_1, \omega_4) = 2.5$, while at time ω_9 it is $degt_{nn}(v_1, \omega_9) = 1$.

An analyst may be also interested in the evolution of the ANND over a time interval, like how people's propensity to rent a bike from one popular location and ride to another popular location is changing within a month. We introduce the *average nearest neighbor degree evolution* to define a series of TANND values within a time interval.

Definition 16 (ANND evolution). *The **average nearest neighbor degree evolution** (ANNDE) $degev_{nn}(v, \tau) := \{x_1, x_2, ..., x_m\}$ of a vertex v is a time series of elements $x_i := degt_{nn}(v, \omega)$, with $1 \leq i \leq m$ and $m = \omega_{end} - \omega_{start}$. Each x_i represents the TANND at time*

ω_j, i.e., x_1 at time point ω_{start} and x_m at $\omega_{end} - 1$, for the interval $\tau = [\omega_{start}, \omega_{end})$. The TANND is a special case of the ANNDE: $degev_{nn}(v, \tau) = \{degt_{nn}(v, \omega)\}$, with $\tau = [\omega_i, \omega_{i+1})$ as an interval with a single time point.

For example, the ANNDE of v_1 in the interval $\tau = [\omega_1, \omega_5)$ is $degev_{nn}(v_1, \tau) = \{1, 1, 2.67, 2\}$. For our small example graph, the ANND remains quite small in this interval, which means that the popularity of the neighbours of v_1 does not increase much. Figure 5.4a shows the resulting ANNDE time series of a selected rental station for the real world bike-sharing graph we are using in our evaluation in Section 5.5. One can see that the tendency that rentals happen between popular stations are high during the summer months.

5.2.3 Graph-centric temporal degree metrics

After looking at metrics for individual vertices of a graph, we now develop metrics that concern an entire graph. Several metrics have already been defined for aggregating all vertices of a static graph, such as the *minimum*, *maximum*, and *average degree*.

Definition 17 (Min/max/avg degree of a graph [125])**.** *The **minimum**, **maximum**, and **average degree** of a static graph G are defined as the minimum, maximum, and average value of all vertex degrees $deg(v)$ for all $v \in V$. It holds:*

$$deg_{min}(G) := min\{deg(v) | v \in V\}, \tag{5.7}$$

$$deg_{max}(G) := max\{deg(v) | v \in V\}, \tag{5.8}$$

$$deg_{avg}(G) := \frac{1}{|V|} \sum_{v \in V} deg(v), \tag{5.9}$$

with $deg_{min}(G) \leq deg_{avg}(G) \leq deg_{max}(G)$.

For the example graph in Figure 5.2, the minimum, maximum, and average degrees are $deg_{min}(G) = 3$, $deg_{max}(G) = 7$ and $deg_{avg}(G) = 5.34$.

With the evolution of a graph, any aggregated graph metric can change over time. We therefore define the *minimum*, *maximum*, and *average temporal degree* as an aggregated value of all vertices $V(\omega)$ in a temporal graph at time ω.

Definition 18 (Min/max/avg temporal degree)**.** *The **minimum**, **maximum** and **average temporal degree** of a temporal graph G are the minimum, maximum and average values of all temporal vertex degrees at time ω. With $V(\omega)$ as the set of vertices at time ω it holds:*

$$degt_{min}(G, \omega) := min\{degt(v, \omega) | v \in V(\omega)\}, \tag{5.10}$$

$$degt_{max}(G, \omega) := max\{degt(v, \omega) | v \in V(\omega)\}, \tag{5.11}$$

$$degt_{avg}(G, \omega) := \frac{1}{|V(\omega)|} \sum_{v \in V(\omega)} degt(v, \omega), \tag{5.12}$$

with $degt_{min}(G, \omega) \leq degt_{avg}(G, \omega) \leq degt_{max}(G, \omega)$.

For the example graph in Figure 5.2, at time ω_4, the aggregated degrees are $degt_{min}(G, \omega_4) = 1$, $degt_{max}(G, \omega_4) = 5$ and $degt_{avg}(G, \omega_4) = 2.67$.

Degree range. The minimum degree reveals the smallest set of connections of a graph's vertices, whereas the maximum degree gives a measure of the most connections an vertex has in the graph. The difference between the minimum and maximum degree of any vertex in a graph is called the *degree range* [125]. It provides a measure of the heterogeneity (or gap) between the connectivity of the most and the least connected vertices in a graph [125].

Definition 19 (Degree range [125])**.** *The **degree range** of a static graph $G = (V, E)$, denoted as $deg_r(G)$, is the difference between the maximum and minimum degree:*

$$deg_r(G) = deg_{max}(G) - deg_{min}(G). \tag{5.13}$$

From a static view on the example graph of Figure 5.2, the degree range is $deg_r(G) = 7 - 3 = 4$, which suggests that it has a high inequality related to connectivity. Now considering a temporal graph, the *temporal degree range* provides information about the degree range of a graph at a specific point in time.

Definition 20 (Temporal degree range)**.** *The **temporal degree range** $degt_r(G, \omega)$ of a temporal graph G at time ω is defined as the difference between the maximum and minimum temporal degree:*

$$degt_r(G, \omega) = degt_{max}(G, \omega) - degt_{min}(G, \omega). \tag{5.14}$$

With respect to the example graph from Figure 5.2, the temporal degree range at time ω_4 is $degt_r(G, \omega_4) = 5 - 1 = 4$, which is equal to the static metric, while at times ω_1, ω_6 and ω_{10}, the temporal degree range is $degt_r(G, \omega_1) = degt_r(G, \omega_6) = degt_r(G, \omega_{10}) = 1$. Thus, as the graph evolves, the degree range changes as well.

To obtain any changes of the degree range over a defined time interval, we introduce the *degree range evolution* that defines a series of temporal degree range values for all time points in a given interval.

Definition 21 (Degree range evolution)**.** *The **degree range evolution** $degev_r(G, \tau) := \{x_1, x_2, ..., x_m\}$ of a temporal graph G is a time series of elements $x_i := degt_r(G, \omega)$, with $1 \leq i \leq m$ and $m = \omega_{end} - \omega_{start}$. Each x_i represents the temporal degree range at time ω_j, i.e., x_1 at time point ω_{start} and x_m at $\omega_{end} - 1$, for the interval $\tau = [\omega_{start}, \omega_{end})$. The temporal degree range is a special case of the degree range evolution: $degev_r(G, \tau) := \{degt_r(G, \omega)\}$, with $\tau = [\omega_i, \omega_{i+1})$ as an interval with a single time point.*

For the example graph of Figure 5.2, the degree range evolution for $\tau = [\omega_0, \omega_7)$ is $degev_r(G, \tau) = \{0, 1, 2, 5, 4, 3, 1\}$, which shows a changing gap of connectivity in this interval. Figure 5.4a shows the time series of the degree range evolution for the real world bike sharing graph we are using in our evaluations. One can see that the value is below 2 over the whole year which indicates a low inequality of rentals between all rental stations.

Degree variance. Besides the simple metric of range, Snijders introduced the more complex metric called *degree variance* of a graph [194], which involves its average degree to characterize the *heterogeneity* in connectivity across nodes. This metric reveals information about the spread of both well-connected and not so well-connected vertices in a graph. It is formally defined as follows:

Definition 22 (Degree variance [125]). *The **degree variance** $deg_v(G)$ of a graph G is defined as the sum of the square of the difference between each vertex degree $deg(v)$ and the average degree of the graph $deg_{avg}(G)$, divided by the total number of vertices $|V|$:*

$$deg_v(G) := \frac{\sum_i (deg(v) - deg_{avg}(G))^2}{|V|}. \tag{5.15}$$

This metric quantifies the extent to which there are differences in the connectivity of the vertices in a graph. High differences in connectivity mean high variance; if all node degrees are the same then the degree variance is zero. If the example graph in Figure 5.2 is considered static it has a degree variance of $deg_v(G) = 2.89$.

For temporal graphs, the degree of vertices can change over time, and so can the average degree as well as the number of vertices. Therefore, we formally define the *temporal degree variance* as follows:

Definition 23 (Temporal degree variance). *The **temporal degree variance**, $degt_v(G, \omega)$, of a temporal graph G is defined as the sum of the square of the difference between each temporal vertex degree $degt(v, \omega)$ and the temporal average degree of the graph $degt_{avg}(G, \omega)$ at time ω, divided by the total number of vertices $|V(\omega)|$ at that time:*

$$degt_v(G, \omega) := \frac{\sum_i (degt(v, \omega) - degt_{avg}(G, \omega))^2}{|V(\omega)|}. \tag{5.16}$$

Considering the example graph in Figure 5.2 at ω_4, the temporal degree variance is $degt_v(G, \omega_4) = 2.89$, which is equal to the static value since the inequality of connectivity is the same for this small example. In contrast, at time ω_1, the temporal degree variance is $degt_v(G, \omega_1) = 0.22$ since there is a quite high equality of connectivity at this time. To evaluate whether and how the degree variance changes in a given time interval, i.e., if the inequality of degrees in a graph decreases or increases over time, or if it retains a similar value, we define the *degree variance evolution*.

Definition 24 (Degree variance evolution). *The **degree variance evolution** $degev_v(G, \tau) := \{x_1, x_2, ..., x_m\}$ of a temporal graph G is a time series of elements $x_i := degt_v(G, \omega)$, with $1 \leq i \leq m$ and $m = \omega_{end} - \omega_{start}$. Each x_i represents the temporal degree variance at time ω_j, i.e., x_1 at time point ω_{start} and x_m at $\omega_{end} - 1$, for the interval $\tau = [\omega_{start}, \omega_{end})$. Further, the temporal degree variance is a special case of the degree variance evolution: $degev_v(G, \tau) = \{degt_v(G, \omega_i)\}$ with $\tau = [\omega_i, \omega_{i+1})$ as an interval with a single time point.*

The degree variance evolution of vertex v_1 in the example graph of Figure 5.2, for time interval $\tau = [\omega_0, \omega_5)$, is the series: $degev_v(G, \tau) = \{0, 0.22, 0.89, 2.89, 2.89\}$. The degree variance increases over time in this example, which indicates a growth of the inequality of the vertex' connectivity. With regard to the real world bike sharing graph, Figure 5.4b shows the degree variance evolution of the temporal graph. The inequality of the rental stations' utilization is low over the whole year but reaches its lowest values in the winter months.

5.3 Degree evolution algorithm

We now describe a baseline algorithm that calculates the degree evolution (see Definition 12) for all vertices in a temporal graph.

We assume that the input is a temporal graph $G = (V, E)$ including a set of temporal vertices V and temporal edges E according to the TPGM model described in Section 5.2.1, where the degree type $\Psi = \{in, out, both\}$ is given as configuration parameter. The output of the algorithm is a time series representing the degree evolution for each vertex, where we reduce the size of the result by merging succeeding time points without a degree change into intervals. These intervals are tuples $\langle v_{id}, \tau_i, degt(v_{id}, \omega_j)\rangle$, where v_{id} is a vertex identifier, τ_i is the interval in which the degree is valid without interruption, and $degt(v_{id}, \omega_i)$ the constant temporal degree of v_{id} for any time point ω_j of the interval τ_i. We split the algorithm into five steps which we described next.

(1) Vertex mapping. For each vertex $v \in V$ we extract the vertex identifier and its time interval into a tuple $\langle v_{id}, \omega_{start}, \omega_{end}\rangle$. This tuple is later used as input of step (5). This step can be skipped if the vertex times are not of relevance. Considering our example graph in Figure 5.2, each of the graph's vertices $V = \{v_1, v_2, v_3\}$ is mapped to a tuple, resulting in a set of three tuples $\langle v_1, 0, \infty\rangle$, $\langle v_2, -\infty, \infty\rangle$ and $\langle v_3, 0, 11\rangle$[2].

(2) Edge mapping. For each edge $e \in E$ we extract the required vertex identifiers and the edge's time interval into one or two tuples $\langle v_{id}, \omega_{start}, \omega_{end}\rangle$ depending on the degree type Ψ. For $\Psi = in$, one tuple is created with $v_{id} \leftarrow t_{id}$ (the target vertex identifier), for $\Psi = out$ one tuple is created with $v_{id} \leftarrow s_{id}$ (the source vertex identifier), and for $\Psi = both$ both of these tuples are created. Considering the example graph in Figure 5.2 and $\Psi = out$, each of the graphs edges $E = \{e_1, e_2, ..., e_8\}$ is mapped to one tuple as described above. For example, edge e_4 is mapped to $\langle v_2, 6, 10\rangle$, whereas e_5 is mapped to $\langle v_2, 3, 6\rangle$.

(3) Interval collection. We group the set of tuples $\langle v_{id}, \omega_{start}, \omega_{end}\rangle$ from step (2) by vertex identifier and create a mapping $v_{id} \rightarrow I_{v_{id}} = \{\tau_0, \tau_1, ..., \tau_n\}$ which assigns a unsorted set of edge intervals $I_{v_{id}}$ to the corresponding vertex identifier. For vertex v_1 and $\Psi = out$ of our example, the mapping to the collection of all incident edge intervals is $v_1 \rightarrow I_{v_1} = \{[1, 5), [2, 6), [3, 4)\}$.

(4) Capture degree evolution. For each vertex v and its corresponding unsorted set of (incoming, outgoing, or both) edge intervals created in step (3), a data structure maintaining the rise or fall of the metric at all respective points in time, i.e., when the degree of the vertex changes, is needed. A baseline approach is the maintenance of a typed list holding two types of points in time: the lower interval bounds which indicate a degree rise of 1, and the upper interval bounds which indicate a fall of 1. The space complexity is always $O(n)$ with $n = 2 \cdot |I_{v_{id}}|$, i.e., the number of all time points including duplicates. All points in time can be inserted with a time complexity $O(n)$ ($O(1)$ each), and the list has to be sorted before the iteration which costs $O(n \cdot log(n))$. The degree evolution for this vertex can be created by iterating the list (with $O(n)$) and adding 1 to a aggregate value for all lower interval bounds and -1 for all upper bounds.

An alternative is a Binary Search Tree (BST) [27] T_v. Each node of the tree has a value $\omega \in \Omega$ and a payload $\rho \in \mathbb{Z}$. ω represents a point in time, whereas ρ (initialized with 0) stores an aggregated value indicating the quantity of change (positive or negative) of the degree at this specific time ω compared to the aggregated value of the evolution until this point in time. For a left-close right-open interval $\tau = [\omega_{start}, \omega_{end})$, the payload ρ of node ω_{start} is increased by 1, whereas ρ of ω_{end} is decreased by 1. Further, the left child node ω_l of a parent node ω_p has a value $\omega_l < \omega_p$ and the right child node ω_r has a value $\omega_r > \omega_p$, respectively. The worst case space complexity is $O(n)$, too, but having

[2]Note that we use integers for time points to improve readability.

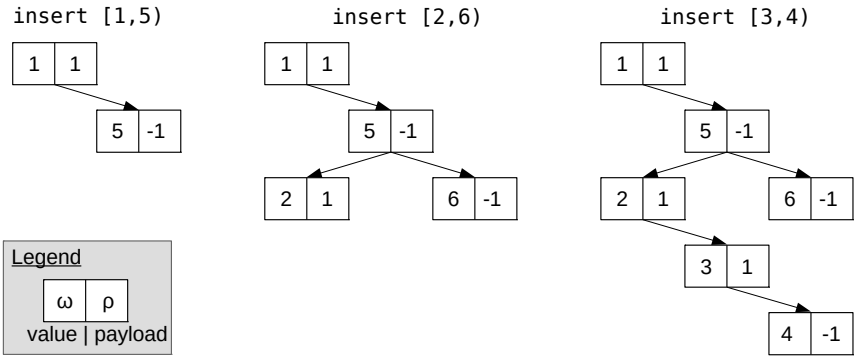


Figure 5.5: Degree tree building for vertex v_1 and $\Psi = out$.

n without duplicate time points. The time complexity of inserting a node in this tree is $O(log(n))$ on average ($O(n)$ if all time points are different). The random insertion of points in time, while keeping the tree sorted, and the lower memory requirements by avoiding duplicated points in time, is our reason for choosing the BST, which will be called *degree tree* in the following. Thus, the output of this step (4) is a mapping $v_{id} \rightarrow T_v$ that assigns a degree tree to its corresponding vertex identifier.

If we again consider v_1 in our example, the building of the degree tree T_{v_1} assuming $\Psi = out$ is shown in Figure 5.5. Inserting the interval $[1, 5)$ first inserts a node with value $\omega = 1$ and payload $\rho = 1$, and then a node with $\omega = 5$ and payload $\rho = -1$. For the subsequent two intervals, four additional nodes are added. A degree tree with six nodes is the result, as shown on the right side.

(5) Tree traversal and result collection For each vertex, we now have a degree tree T_v that represents the degree evolution of this vertex for the degree type Ψ, and the lower and upper bounds of the vertex' validity interval, ω_{start} and ω_{end}. If the validity of the vertices can be neglected, a default minimum and maximum time point can be used as initial values. Each degree tree is now traversed using Depth First Search (DFS) [200] and in-order traversal (LNR) starting at the root node to obtain an ascending order of points in time. Algorithm 1 outlines this step.

The algorithm starts by traversing the tree T_v in line 5 with the recursive function InOrderDFS (lines 8 to 11). Function ProcessNode describes the logic of a node visit, where we first handle the special case of an vertex lower interval bound that is equal to the value of first visited node of the tree (lines 13 to 15). For every following visited node, the resulting temporal degree tuple is collected in line 17 if payload $\rho \neq 0$.

Next, to get the degree for the subsequent interval, the payload ρ is first added to d (line 18), and second the time point ω is remembered as lower interval bound for the next interval (line 19). After all nodes of the tree are visited, we check for a remaining time interval from the last time point ω_{last} to the vertex upper interval bound ω_{max} and collect a last tuple with $d = 0$ accordingly (line 7). The final algorithm output is a series of tuples $\langle v_{id}, \tau, degt(v_{id}, \omega)\rangle$, with $degt(v_{id}, \omega)$ as constant temporal degree for all time points $\omega \in \tau$, that were collected by both *collect()* calls (lines 7 and 17).

For a better understanding, we exemplary go through Algorithm 1 by using the degree tree T_{v_1} of vertex v_1, shown on the right side in Figure 5.5, as input. Remember this is the representation of the outdegree of v_1. In addition, from step (1), the algorithm gets the lower bound $\omega_{start} = 0$ and upper bound $\omega_{end} = \infty$ of the vertex interval as input

Algorithm 1: Tree traversal and result collection

```
   Data: T_v, v_id, ω_start, ω_end;                          /* Input data */
1  ω_last ← ω_start ;                          /* ω_start = −∞ if not given */
2  ω_max ← ω_end ;                              /* ω_end = ∞ if not given */
3  d ← 0 ;                                    /* Initialize degree with 0 */
4  Function Main():
5  |  InOrderDFS(T_v);          /* Traverse the tree with in-order DFS */
6  |  if ω_last < ω_max then       /* Check for last remaining interval */
7  |  |  collect(⟨v_id, [ω_last, ω_max), d⟩);          /* Collect tuple for last
   |  |   interval */
8  Function InOrderDFS(tree):
9  |  if tree.left ≠ null then InOrderDFS(tree.left);
10 |  ProcessNode(tree.value,tree.payload);
11 |  if tree.right ≠ null then InOrderDFS(tree.right);
12 Function ProcessNode(ω, ρ):
13 |  if ω_last == ω then                          /* Check first node visit */
14 |  |  d ← d + ρ;                                /* Add payload to degree */
15 |  |  return ;                                        /* Leave function */
16 |  if ρ ≠ 0 then                          /* Check if the degree changes */
17 |  |  collect(⟨v_id, [ω_last, ω), d⟩) ;                    /* Collect tuple */
18 |  |  d ← d + ρ ;                        /* Add degree change to degree */
19 |  |  ω_last ← ω;                          /* Remember ω for next call */
```

parameters to initialize ω_{last} and ω_{max}. During the in-order traversal of the DFS, function ProcessNode(ω, ρ) is called first with the arguments $(1, 1)$ (value,payload), followed by $(2, 1)$, $(3, 1)$, $(4, -1)$, $(5, -1)$ and $(6, -1)$.

According to the first tuple, the interval $[0, 1)$ is defined and collected as part of the first resulting temporal degree tuple $\langle v_1, [0, 1), 0\rangle$ afterwards (line 17). Then, the payload 1 is added to the degree value d (line 18) and the timestamp value 1 is remembered in variable ω_{last} (line 19). In the next function call with input tuple $(2, 1)$, an interval $\tau \leftarrow [1, 2)$ is defined and collected together with the current degree value of d which is 1. The collected result tuple is thus $\langle v_1, [1, 2), 1\rangle$. Again, the degree value is updated by the payload and the timestamp is remembered. For the remaining four input tuples $(3, 1)$, $(4, -1)$, $(5, -1)$ and $(6, -1)$ will be the following result tuples collected: $\langle v_1, [2, 3), 2\rangle$, $\langle v_1, [3, 4), 3\rangle$, $\langle v_1, [4, 5), 2\rangle$ and $\langle v_1, [5, 6), 1\rangle$.

To collect also the remaining interval from 6 to ∞, the condition (line 7) checks whether the largest timestamp in the tree (ω_{last}) is smaller than the maximum timestamp ($\omega_{max} = \infty$). Since this is true in our case, we define the remaining interval $\tau = [6, \infty)$ and collect the output tuple $\langle v_1, [6, \infty), 0\rangle$ which states that the degree of v_1 is 0 for the interval $[6, \infty)$. The result of this final step is a compact representation of the *degree evolution* of the outdegree of vertex v_1 as defined by Definition 12: $degev^+(v_1, [0, \infty)) = \{\langle 0, [0, 1)\rangle, \langle 1, [1, 2)\rangle,$
$\langle 2, [2, 3)\rangle, \langle 3, [3, 4)\rangle, \langle 2, [4, 5)\rangle \langle 1, [5, 6)\rangle \langle 0, [6, \infty)\rangle\}$.

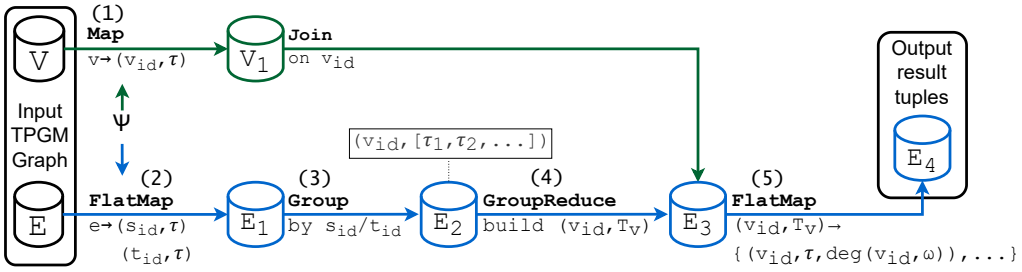


Figure 5.6: Implementation details of the Degree Evolution-Operator.

5.4 Distributed implementation

The ability to process very large graphs efficiently is often a limitation of existing graph processing systems [187], requiring partitioning of large graphs and distributed processing for example of analytical tasks. There are distributed graph processing systems, such as Tegra [100] based on Apache Spark [225], or Gradoop [175, 176] which uses Apache Flink [37]. An analytical operator in Gradoop is a smart combination of Flink transformations. A Flink transformation, e.g., *map*, *flatMap* and *join*, is a processing unit that can be applied in parallel on a distributed Flink DataSet. A DataSet represents a distributed collection of elements of the same type in Apache Flink. Its tuples are distributed among all nodes of a cluster according to a partitioning strategy. We use this operator concept for our distributed implementation of the algorithm described in Section 5.3.

Figure 5.6 shows an architectural sketch of a *Degree Evolution-Operator*[3] as a Directed Acyclic Graph (DAG) representing multiple Flink transformations that are applied on the input graph DataSets: V with $v_i = \langle v_{id}, \tau \rangle$ and E with $e_i = \langle e_{id}, s_{id}, t_{id}, \tau \rangle$. The enumeration of the data flow follows the algorithm steps given in Section 5.3.

First, in step (1), each vertex of the input vertex DataSet V, is mapped to a minimal representation holding the vertex identifier and the bounds of the vertex' time interval. The resulting DataSet is named V_1 in the figure. If the temporal information of the vertices can be neglected, this step can be skipped and default min/max timestamps can be used as input to step (5), which avoids the later described distributed join. Then, we apply a *FlatMap* transformation, step (2), to the edge DataSet E that is configured by the degree type ($\Psi \in \{in, out, both\}$) as selected by the user. According to the degree type, one or two tuples of the format $\langle v_{id}, \omega_{start}, \omega_{end} \rangle$ are extracted from an input edge tuple (step (2) in Section 5.3). The resulting DataSet is denoted as E_1.

On E_1, we apply a *Group* transformation which groups all entities by the vertex identifier, and creates a set of tuples $\langle \omega_{start}, \omega_{end} \rangle$ for each group. In the figure, this step is marked by (3), whereas the resulting grouped DataSet is denoted as E_2. Due to the grouping, E_2 is partitioned by the vertex identifier. For each group, we apply a *GroupReduce* transformation in step (4) which calls a user-defined function for each group. This function receives the whole group at once and produces a mapping $v_{id} \rightarrow T_v$ assigning a degree tree to its corresponding vertex identifier, represented as a tuple

[3]The operator code is open-source: https://github.com/dbs-leipzig/gradoop/tree/develop/gradoop-temporal/src/main/java/org/gradoop/temporal/model/impl/operators/metric.

	$\|V\|$	$\|E\|$	Size (GB)	$\sum \|degev()\|$
LDBC SF1	3.2 M	17.3 M	4.2	30.6 M
LDBC SF10	30.0 M	176.6 M	42.3	319.6 M
LDBC SF100	282.6 M	1.77 B	421.9	3.18 B
Citi Bike	1174	97.5 M	22.6	381.0 M
Stackoverflow	462.9 M	664.8 M	199.0	1.3 B

Table 5.1: Dataset statistics, including their sizes on HDFS and number of result set tuples for $\Psi = both$, i. e., $\sum_{i=1}^{|V|} |degev(v_i)|$. For example, 3.18B tuples result for the LDBC dataset with SF 100.

$\langle v_{id}, T_v \rangle$. The resulting tuples are part of DataSet E_3, which is partitioned by the vertex identifier.

Now, each tuple of V_1 needs to be joined by the vertex identifier to it's corresponding degree tree tuple of DataSet E_3 to extend it with the interval bounds of the vertex. As said before, this step can be optionally skipped. As a result of the join, the DataSet E_3 consists of tuples $\langle v_{id}, T_v, \omega_{start}, \omega_{end} \rangle$. As a last step, annotated with a (5), a *FlatMap* transformation is applied on DataSet E_3 where its internal logic implements the tree traversal and result collection process defined in Algorithm 1. For each input tuple, the transformation produces multiple (at least one) result tuples in the form $\langle v_{id}, \tau, degt(v_{id}, \omega) \rangle$, describing the constant temporal degree (see Definition 11) of vertex $v \in V$ (identified by v_{id}) for the whole interval τ. The resulting DataSet is named E_4.

5.5 Experimental Evaluation

We now evaluate the runtime and scalability of the temporal degree operator we discussed in Section 5.4 with respect to increasing data set and cluster sizes. We ran all experiments on a cluster with 16 worker nodes connected via 1 GBit Ethernet, where each worker consists of a E5-2430 6(12) 2.5 Ghz CPU, 48 GB RAM, two 4 TB SATA disks, and running openSuse 13.2, Hadoop 2.7.3 and Flink 1.9.0. On a worker node, a Flink Task Manager [37] is configured with 6 task slots and 40GB memory.

We use three datasets for the evaluation, referred to as *LDBC* [97] (a synthetic social network in three scale factors: 1, 10 and 100), *citibike* [129] and *stackoverflow* [196] (both real-world data). In Figure 5.4 we show example time series of four evolution metrics for the citibike dataset. Each graph is stored distributed using the Hadoop Distributed File System (HDFS) by hash partitioning as two datasets V and E. Table 5.1 shows statistics of the three datasets with the different scaling factors (SF) for LDBC. Each experiment includes reading the graph dataset from the HDFS, executing the specific workflow, and finally writing all results back to the HDFS. We ran each experiment five times and report average runtimes.

Impact of dataset size. Figures 5.7 and 5.8 show the impact of the dataset size to the operator runtime with full parallelism of 16 workers with respect to different degree types $\Psi \in \{in, out, both\}$. While Figure 5.7 shows the actual runtime in seconds for all three dataset sizes, Figure 5.8 visualizes the factor by which the runtime has increased compared to the runtime of the LDBC SF1 dataset. For example, the runtime for the LDBC SF1 dataset for $\Psi = both$ is only 23.3 seconds, for LDBC SF10 164.6 seconds (factor

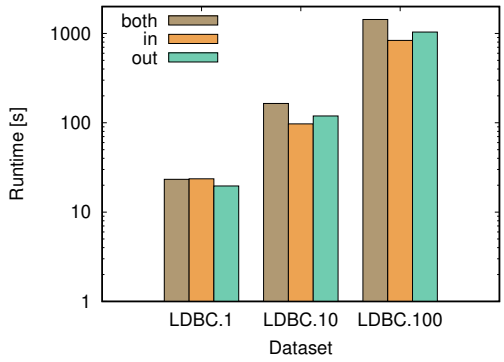


Figure 5.7: Runtimes for linearly growing dataset sizes.

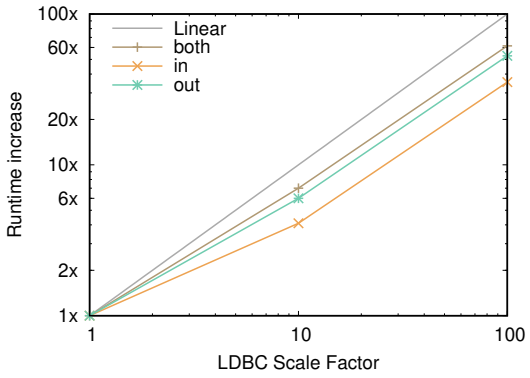


Figure 5.8: Factor of runtime increase for linearly growing dataset sizes.

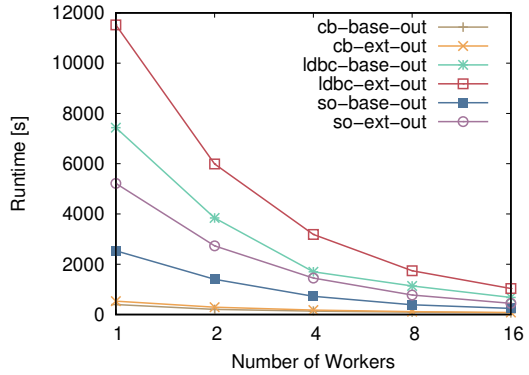


Figure 5.9: Runtimes for #workers with $\Psi = out$.

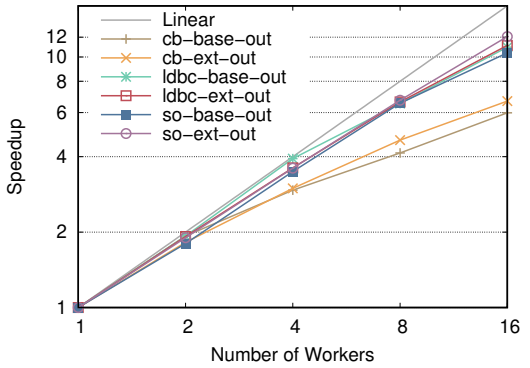


Figure 5.10: Speedup of algorithm for $\Psi = out$.

7 higher compared to LDBC SF1) and for LDBC SF100 1433.6 seconds (factor 61). The best result is given by degree type $\Psi = in$, where the runtimes of LDBC SF100 are only 35.4 times larger compared to LDBC SF1, although the dataset is 100 times larger. From LDBC SF10 to LDBC SF100 the runtimes of all three degree types rise equally.

The results, specifically Figure 5.8, show that a linear increase of the dataset size leads to only a sublinear increase in the running time for a constant graph structure. Further, the runtimes of $\Psi = both$ are always higher compared to the others which is due to the double amount of collected tuples in step (2), as we discussed in Section 5.4.

Impact of worker count. We next examine the runtime and scalability of the algorithm for all datasets. In addition, the effect of excluding the vertex time information as described in Section 5.3 is evaluated. Without using the vertex time, the complete step (1) and the expensive join after step (4) can be avoided (see Section 5.4). In the following, we refer to an execution without vertex time as *base* and *extended* for the full algorithm. The results in Figure 5.9 show that the mentioned higher complexity has a significant impact on the running time. For example, the runtime on a single machine for the citibike dataset is 397.6 seconds (base) and 533.6 seconds (extended), which means an increase of 34.2%. For the stackoverflow dataset, the execution takes 2,536 seconds (base) and 5,216 seconds (extended), which means almost doubling the runtime on a single machine.

The more workers are added, the smaller the runtime and the difference between the

two algorithm variants, which can be seen in Figure 5.10. With the citibike dataset, we can see that the runtimes on a single machine are already low and that only a moderate improvement can be achieved through horizontal scaling of resources. For this dataset, we reach a speedup of about 6.7 for 16 machines using the extended variant, while for the LDBC SF100 and stackoverflow datasets, we achieve a speedup of up to 11.1 and 12.07, respectively.

5.6 Conclusion

Most graphs that model real-world entities and their relationships are dynamic, where edges and vertices can be valid for only a certain period of time. One simple but often used centrality measure is the degree centrality using a vertex' degree to judge it's popularity in a network.

We show in this work that it is necessary to determine a vertex degree over time, to know exactly *when* a node has which degree and *how long* this value is valid and in which quantity it does change over time. We therefore provide temporal extensions to the vertex degree metric itself, its aggregations and others based on it, namely the degree range, the degree variance and the ANND, and define them formally. We further describe an algorithm to calculate the newly introduced degree evolution for all vertices of a temporal graph. We implemented the algorithm as a graph analysis operator in Gradoop, an open-source distributed graph analysis system.

We evaluated runtimes and scalability of the operator on a cluster with 16 machines to determine the impact of different datasets and sizes. In summary, we have shown that a linear increase in the dataset size leads to only a sublinear increase in runtime of our algorithm. We also showed that the operator scales well by increasing the number of machines. Speedup values between 10 and 12 were achieved on 16 machines using the two largest datasets.

Part III

Continuous Querying

6

The Fusion of Graph and Time-Series Data

This work is a summarized view on the results of a one-year cooperation between Oracle Labs and the University of Leipzig under the grant name "Temporal Property Graphs as Organizing Principles" [76]. The goal was to research the organization of relationships within multi-dimensional time-series data, such as sensor data from the IoT area. We showed in this project that temporal property graphs with some extensions are a prime candidate for this organizational task that combines the strengths of both data models (graph and time-series). The outcome of the cooperation includes three achievements, which are summarized in this section: 1) a bitemporal property graph model as an extension of the already introduced TPGM, 2) a temporal graph query language based on Oracle's PGQL language, and 3) a conception of a continuous graph-based event detection approach.

The contents of this section were published on arXiv under the title *Bitemporal Property Graphs to Organize Evolving Systems* [173].

6.1 Introduction

Time series data are omnipresent. From share prices, population trends, price indices, weather data or sensor readings from complex machines, time series data can be multi-dimensional and generated discretely in finite time intervals. The inherent temporal dimension of time series data provides a rich source of information, enabling the analysis of sequential events and dependencies, essential for understanding and predicting various real-world processes. However, a prominent challenge is the organization of such multi-dimensional time series data, e.g., to connect correlating sensor data from the IoT area. Even though time series data has very high information content, the data structure offers no possibility of depicting or describing the relationships between entities that produce time series. For example, an aircraft, like the one in Figure 6.1, is a complex system of many bigger and smaller components, many of which are equipped with various sensors that continuously capture data to determine its current status. One component (also denoted as an *asset*) is often a complex system of smaller components, e.g., the airplane turbine, which is again equipped with sensors.

Imagine that each sensor in such a complex system delivers a time series of values,

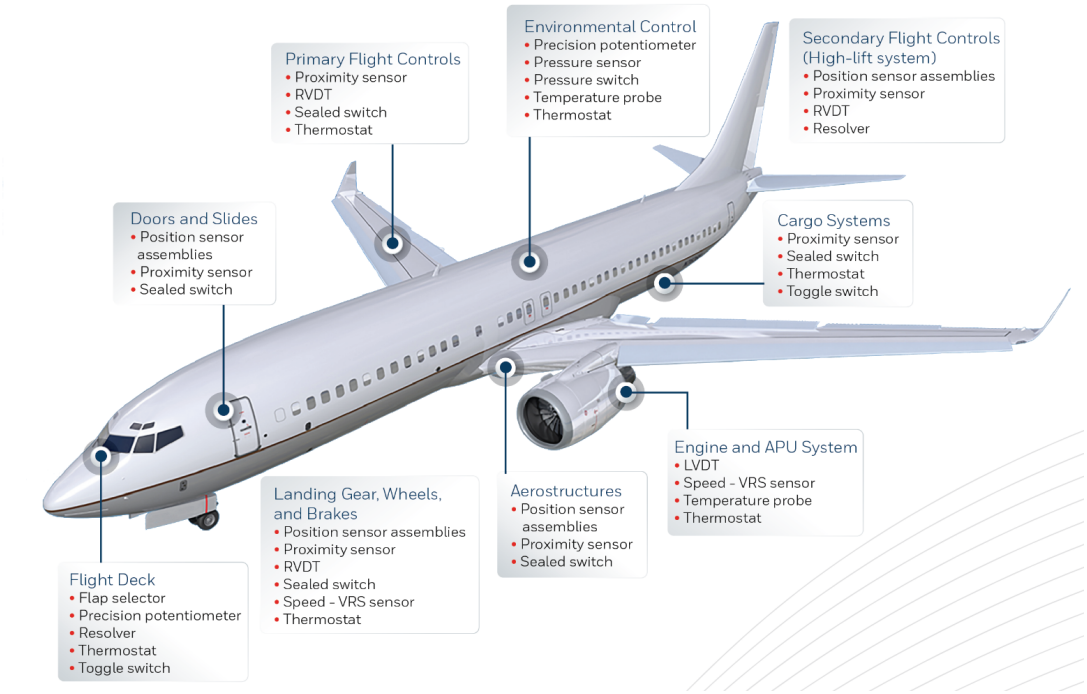


Figure 6.1: Sensors of an airplane [92].

e.g., temperatures, rotational speeds, centrifugal forces, accelerations, etc. To model the relationships between sensors and components themselves, as well as sensors and components among each other, a data structure is required that can model such a complex, heterogeneous network and allow structural and content changes to the network and store precisely when these changes took place. Having such a network (or graph), one can query for these relationships and find correlations that might be hidden by looking at the time series isolated from each other.

We figured out that temporal property graphs are the prime candidates for this organizational task that combines the strengths of both data models (graph and time series). Several requirements emerged for this technology combination, including: 1) the need for a rich graph data model with full auditing of the graph's evolution - in the observed real world as well as in the graph database, 2) the need for a declarative query language to match patterns in a changing graph, and 3) the need for an event detection mechanism through a continuous evaluation of registered patterns against graph changes.

The contributions of this collaborative project addressing the challenges C1, C2, C3 and C10 from Section 1.1 and can be summarized as follows:

- the **TPGM$^+$**, a bitemporal property graph model based on the TPGM with a focused support of property changes;
- the **T-PGQL**, a temporal graph query language with similar features as *Temporal-GDL* but based on Oracle's PGQL language,
- and Continuous Graph Notifications **CGN**, a mechanism to continuously notify about query result updates.

Section 6.2 introduces the TPGM$^+$, whereas Section 6.3 presents T-PGQL with many example queries. The Continuous Graph Notification (CGN), a technology for continuously

evaluating registered T-PGQL queries for event detection, is introduced in Section 6.4. We summarize the work and give an outlook on ongoing work in Section 6.5.

6.2 TPGM⁺: A TPGM EXTENSION WITH TEMPORAL PROPERTIES

A property graph (see Section 2.1) is characterized by *vertices* and directed *edges*, type *labels* and *properties*. The latter are represented as key-value pairs and describe the entity or relationship in more detail, e.g., `name:Christopher` or `city:Leipzig`. Based on the PGM, we previously introduced the TPGM and formally defined it in Section 3.3. One downside of the TPGM is the limited support for property changes. If a property of a vertex or edge changes, i.e., it is created updated or removed, a logical copy of the vertex or edge is created that stores the updated state and the information about the time of the update. All properties that are not affected by the changes stay unchanged and exist as overhead in this new version. Thus, high frequent changes of properties and their values thus result in a huge amount of duplicated elements.

To overcome this weakness, we defined the $TPGM^+$, an extended version of the TPGM that adds bitemporal modeling to the layer of properties. For each $TPGM^+$ graph, two linear ordered discrete time domains Ω exist: Ω^{tx} for the transaction time and Ω^{val} for the valid-time. Each instant in time is a time point ω_i. The linear ordering is described as $\omega_i < \omega_{i+1}$, i.e., ω_i happened before ω_{i+1}. Per time domain, each vertex, edge, **and property value** has an associated time-period τ. Given $\omega_s, \omega_e \in \Omega$, a time-period $\tau = [\omega_s, \omega_e)$ is defined as a close-open time interval starting at and including ω_s and ending at but excluding ω_e. The time points of τ are thus a set $\{t \mid t \in \Omega \wedge \omega_s \leq t < \omega_e\}$. The length or size of a period τ is defined by $len(\tau)$. Default values for lower and upper bounds are $\omega_{min} = -\infty$ and $\omega_{max} = \infty$.

Further, we defined several integrity constraints of a $TPGM^+$ graph, to ensure the consistency of the graph at each point in time. Each constraint is valid per time domain.

1. *Unique vertices, edges and properties.* At every point in time, vertices, edges, and properties are unique, i.e., exist at most once. The uniqueness constraint of vertices and edges is a combined key of the element's identifier and the end timestamp. The uniqueness constraint of a property is its name.
2. *Referential integrity of edges.* For each edge, the time intervals associated with its source and target vertices must contain the edge's time interval. In other words, an edge can only exist when its incident vertices exist.
3. *Referential integrity of properties.* For a property value, the interval of the vertex/edge must contain the interval of the property value. In other words, a property value can only exist when the corresponding vertex/edge exists.
4. *Constant edges.* Source and target vertices of an edge never change at existence.
5. *Constant types.* Vertex and edge labels, as well as property key names, never change.

Figure 6.2 shows an example of a $TPGM^+$ graph that demonstrates the evolution of property values. The temporal graph shows two `Sensor` vertices that are connected to an `Asset` vertex. The time dimensions on the vertex and edge layer are already known from the TPGM and are for demonstration purposes valid forever (through the interval

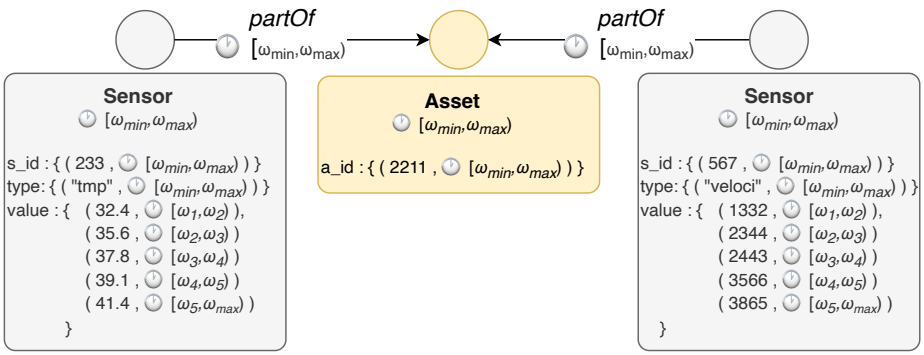


Figure 6.2: Example of a TPGM$^+$ graph with updates of vertex properties. For simplicity, just the valid-time dimension is exemplified.

$[\omega_{min}, \omega_{max}))$. This also holds for rather static property values, like the sensor ID or type. For changing property values, like the measured value of a sensor, we have the time information as an interval attached to each value. For example, sensor 233 captured the value 37.8 at time ω_3, which is valid until the next value was captured at ω_4. This simple extension of the data model now makes it possible to model time series data as a changing property of a node or edge.

6.3 T-PGQL: A Temporal Property Graph Query Language

In Section 3.3, we already introduced a bitemporal graph query language called *TemporalGDL*. The underlying GDL [108] language is rather a research prototype than an industrial language supported by commercial graph databases. In contrast, PGQL [150, 166], Oracle's declarative graph query language, is fully supported by Oracle's graph database. In cooperation with Oracle, we developed *T-PGQL*, an alternative temporal graph query language based on PGQL. It adapts some of the features of TemporalGDL, but also has new features that specifically support the temporality of properties introduced by TPGM$^+$.

We will first look at some preliminaries needed to understand the extensions we made to the graph query language PGQL, which are explained afterward.

6.3.1 Preliminary: PGQL - A (Non-temporal) Graph Query Language

PGQL [150, 166] combines graph pattern matching with SQL-like syntax and functionality and has full-blown support for regular path queries and graph construction. Because its syntax is SQL-like, the language is intuitive to use for existing SQL users. Furthermore, PGQL queries return a result set with variables and bindings, just like in SQL. In this work, we refer to PGQL version 1.3 [151].

Matching graph patterns is one main functionality of PGQL realized by a SELECT query. Similar to SQL, a SELECT query is composed of several clauses, starting with the mandatory `SELECT` clause and `FROM` clause. In PGQL, the `SELECT` clause defines the

returned result entities of the query. Since the result of a SELECT query is always a table, the SELECT clause defines the attributes of the result table. The graph pattern to be matched is defined by the FROM clause that includes one or multiple MATCH clauses. A MATCH clause defines a path- or graph pattern, where a graph pattern is a composition of path patterns. Listing 6.1 shows the syntactic structure of a PGQL SELECT query including the FROM- and MATCH clauses, taken from [151].

```
SelectQuery     ::=  SelectClause
                     FromClause
                     WhereClause?
                     ...
SelectClause    ::= 'SELECT' 'DISTINCT'? ExpAsVar ( ',' ExpAsVar )*
                  | 'SELECT' '*'
FromClause      ::= 'FROM' MatchClause ( ',' MatchClause )*
MatchClause     ::= 'MATCH' ( PathPattern | GraphPattern ) OnClause?
GraphPattern    ::= '(' PathPattern ( ',' PathPattern )* ')'
PathPattern     ::=   SimplePathPattern | ShortestPathPattern | ···
ExpAsVar        ::= ValueExpression ( 'AS' VariableName )?
```

Listing 6.1: Syntax definition of a PGQL Select Query [151].

Path patterns describe topology constraints, where a topology constraint is a composition of one or multiple vertices and edges. A vertex or edge is optionally identified by a variable, i.e., a symbolic name to reference it in other clauses. It is also possible to define one or more label predicates directly in the pattern. A PGQL SELECT query returning all person and movie names that match this pattern can be formulated as follows:

```
1 SELECT p.name, m.movie
2 FROM MATCH (p:Person)-[l:like|dislike]->(m:Movie)
```

For more details we refer to the official documentation of PGQL, available at [150].

6.3.2 Extensions of PGQL to Query TPGM$^+$ Graphs

In current graph query languages patterns (or paths) are searched in the whole available graph database without observance of any evolution. Having a temporal graph modeled by the TPGM$^+$ (see Section 6.2), several new requirements arise for a query language to support temporal features of the model. In this work, we limit to the retrieval of data. The manipulation of data as well as functions for the definition and manipulation of database structures are not considered yet and part of future work. In all subsequent T-PGQL query examples, syntax highlighting was chosen to highlight the extension, showing the original PGQL syntax in blue (e.g., **MATCH**), and the extended T-PGQL syntax green (e.g., **AS OF**).

Access of Temporal Attributes

Our data model tracks changes in a bitemporal model on a level of vertices, edges and properties. We introduce the possibility to add these attributes via projection to the resulting relation and to use these attributes in expressions for selection. We distinguish four different temporal characteristics of a vertex, edge and property: 1) the period itself,

```
TimeIdentifier  ::= 'TX_TIME' | 'VAL_TIME' | 'TX_FROM' | 'TX_TO' | 'VAL_FROM'
    | 'VAL_TO'
Property        ::= Identifier
VarRef          ::= Identifier
PropRef         ::= VarRef '.' Property
ElementTimeRef  ::= VarRef '.' TimeIdentifier
PropertyTimeRef ::= PropRef '.' TimeIdentifier
TimeRef         ::= ElementTimeRef | PropertyTimeRef
```

Listing 6.2: Syntax definition of accessing temporal attributes in T-PGQL.

2) its lower bound timestamp (inclusive), 3) its upper bound timestamp (exclusive) and 4) the length/duration of a period.

The period identifier is defined as `VAL_TIME` for the valid time domain and `TX_TIME` for the transaction time domain. A period is, like in SQL, not a data type but a type definition [120]. The textual representation a period projection is a concatenated result of both period bounds in the form of: `[{from},{to})`. A period type can be used for several predicates, as later described. The identifiers for the period bounds are `VAL_FROM` and `VAL_TO` for the valid time domain and `TX_FROM` and `TX_TO` for the transaction time domain. The result of a period-bound access is a timestamp.

To access these attributes of a vertex or edge, they can be used like a property access by dot notation, e.g., `var1.TX_FROM` or `var2.VAL_TIME`. According to the temporal attributes of a property, the same suffix can be used on a property access expression, like `var1.prop1.TX_FROM` or `var2.prop2.VAL_TIME`. Listing 6.2 shows the respective syntax definition.

For example, the following query returns all available temporal characteristics of person vertices and their name property.

```
SELECT n.TX_FROM, n.TX_TO, n.TX_TIME,
       n.VAL_FROM, n.VAL_TO, n.VAL_TIME,
       n.name.TX_FROM, n.name.TX_TO, n.name.TX_TIME,
       n.name.VAL_FROM, n.name.VAL_TO, n.name.VAL_TIME
FROM MATCH (n:Person) ON student_network
```

Besides the period of a graph element or its property, the length/duration of this period can be queried. We introduce a `LENGTH([unit,]period)` expression which consumes a period access identifier (`period`) as argument and an optional unit (i.e., `YEAR`, `QUARTER`, `MONTH`, `WEEK`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`). If no unit is given, the default unit `MILLISECOND` is used. This expression returns a numerical value that can be used within several expressions, e.g., binary constraints. The following query returns the duration in days of the valid time period of a person's name property for all persons whose valid time exceeds one day.

```
SELECT LENGTH(DAY, n.name.VAL_TIME)
FROM MATCH (n:Person) ON student_network
WHERE LENGTH(DAY, n.name.VAL_TIME) > 1
```

Temporal Filtering and Chronological Pattern Matching

Graphs with a managed valid-time are intended for meeting the requirements of applications that are interested in capturing time periods during which the data is (believed to be) valid in the real world. For each vertex or edge, as well as their properties, a valid-time period is available. As described above, the identifier of this period is `VALID_TIME` and the beginning and ending bounds are `VAL_FROM` and `VAL_TO`. They can be used as a suffix to variable and property access. Analogous to the transaction time attributes, the valid time period returns a new *period* type definition and the bounds return a timestamp type. Latter can be used like regular timestamp attributes, e.g., within binary relations in the `WHERE` clause, which is defined in Listing 6.3. Note that `TemporalExpression` is a temporal extension of this work and not part of the origin PGQL clause.

```
WhereClause        ::= 'WHERE' ValueExpression
ValueExpression    ::= VariableReference
                     | PropertyAccess
                     | ...
                     | TemporalExpression
TemporalExpression ::= ElementTimeRef
                     | PropertyTimeRef
                     | Overlaps | Equals
                     | Precedes | Succeeds
                     | Contains
Overlaps           ::= TimeRef 'OVERLAPS' TimeRef
Equals             ::= TimeRef 'EQUALS' TimeRef
Precedes           ::= TimeRef ('IMMEDIATELY')? 'PRECEDES' TimeRef
Succeeds           ::= TimeRef ('IMMEDIATELY')? 'SUCCEEDS' TimeRef
Contains           ::= TimeRef 'CONTAINS' TimeRef
```

Listing 6.3: Syntax definition of the T-PGQL WHERE clause.

For example, to retrieve all students who studied at a University in Leipzig as of February 15, 2019, one can express the query by accessing the period bounds in predicates of the `WHERE` clause:

```
SELECT n.name
FROM MATCH (n:Person)-[s:studiedAt]->(u:University)
WHERE u.city = 'Leipzig'
    AND s.VAL_FROM <= DATE '2019-02-15'
    AND s.VAL_TO > DATE '2019-02-15'
```

To simplify the expression of such predicates, several language extensions are further defined. For example, we can use one of the period predicates provided in SQL:2011 [120] for expressing conditions involving periods: `CONTAINS`, `OVERLAPS`, `EQUALS`, `(IMMEDIATELY) PRECEDES`, and `(IMMEDIATELY) SUCCEEDS`. All period predicates need two expressions that return a period as arguments, except for `CONTAINS`, which also allows a timestamp as a second argument. The query above can be simplified by using the `CONTAINS` predicate:

```
SELECT n.name
FROM MATCH (n:Person)-[s:studiedAt]->(u:University)
WHERE u.city = 'Leipzig'
    AND s.VAL_TIME CONTAINS DATE '2019-02-15'
```

To retrieve all students of Universities in Leipzig who are matriculated from January 1, 2018, to January 1, 2019, one could formulate the query by using a temporal condition, in our case, the OVERLAPS predicate:

```
SELECT n.name
FROM MATCH (n:Person)-[s:studiedAt]->(u:University)
WHERE u.city = 'Leipzig'
    AND s.VAL_TIME OVERLAPS PERIOD(DATE '2018-01-01', DATE '2019-01-01')
```

In the example, we also show a new period constructor expression (**PERIOD**(ω_1, ω_2)) that allows the creation of a period instance from two timestamps ω_1 and ω_2 with $\omega_1 \leq \omega_2$. The timestamps can be defined by any expression that returns a date or timestamp instance. For example, they can be created through a date or timestamp constructor as in the example or extracted from a graph element or property through a period bound identifier, like shown in the next query.

```
SELECT n.name
FROM MATCH (n:Person)-[s:studiedAt]->(u:University)
WHERE u.city = 'Leipzig'
    AND s.VAL_TIME OVERLAPS PERIOD(DATE '2018-01-01', n.VAL_TO)
```

In addition, the transaction time period of elements and properties can be used in such predicates. The following query returns the name and system-time period for students matriculated in a University located in Leipzig where the information about the matriculation was added to the database after March 1, 2018. The syntax part **FOR TX_TIME ALL** is explained in the next section.

```
SELECT n.name, n.TX_TIME
FROM MATCH (n:Person)-[s:studiedAt]->(u:University) FOR TX_TIME ALL
WHERE u.city = 'Leipzig'
    AND s.TX_FROM >= TIMESTAMP '2018-03-01 00:00:00'
```

The classical graph pattern matching is used to find a matching subgraph in the graph database that matches exactly with the defined query pattern. In static languages, a query pattern has no information about the chronological ordering of the given entities and relationships. For example, the following pattern includes no information about when the likes happened or if one like happened before the other of if they happened at the same time:

```
(p1:Person)-[l1:likes]->(p2:Person)-[l2:likes]->(p3:Person)
```

To overcome this lack, the above introduced temporal predicates can be used to enrich such patterns with temporal information. The following code exemplifies that:

```
SELECT p1.name, p2.name, p3.name
FROM MATCH (p1:Person)-[l1:likes]->(p2:Person)-[l2:likes]->(p3:Person)
WHERE l1.VAL_TIME PRECEDES l2.VAL_TIME
```

Here we add a constraint, that the like between p1 and p2 must happen before the like between p2 and p3. These kinds of predicates can thus be used to define a temporal ordering in a path pattern.

Graph Snapshot Retrieval and Historical Pattern Matching

T-PGQL can be used to find matching subgraphs in a specific state of the temporal graph concerning the transaction time domain. This state can be 1) a snapshot at a defined timestamp or 2) all changes of a given period. Former represents a valid snapshot graph without multiple versions of a single instance. The transaction time dimension is reduced to a single point in time. Latter is again a temporal property graph that can have multiple versions of a single instance. The transaction time dimension is reduced to a range in time.

To search for a defined pattern using this kind of time traveling, we extended PGQL's `FROM` clause with statements similar to the SQL extension for temporal tables. The name of the transaction time period definition is fixed to **`TX_TIME`**. To query the historical data, the clause **`FOR TX_TIME`** `{predicate}` has to be used directly after a MATCH clause (see Section 6.3.1). To define the timestamp or period to query for, we provide four predicates as syntactic extensions, as stated in Listing 6.4.

```
GraphMatch          ::= 'MATCH' PathPattern OnClause? SysTimeCond
SysTimeCond         ::= 'FOR' 'TX_TIME' ( AsOf | FromTo
                                        | BetweenAnd | 'ALL')
AsOf                ::= 'AS' 'OF' TimeRef
FromTo              ::= 'FROM' TimeRef 'TO' TimeRef
BetweenAnd          ::= 'BETWEEN' TimeRef 'AND' TimeRef
```

Listing 6.4: Syntax definition of the T-PGQL MATCH clause.

The argument `TimeRef` could be any expression returning a timestamp attribute, i.e., a date or timestamp constructor (e.g., `TIMESTAMP('2020-01-01')`), current timestamp expression (`CURRENT_TIMESTAMP`) or access expressions of temporal attributes for vertices, edges or properties. If the **`FOR TX_TIME`** clause is not used, the result will show the current data, as if one had specified **`FOR TX_TIME AS OF`** `CURRENT_TIMESTAMP`. Thus, the following two queries are equal:

```
1 SELECT n.name
2 FROM MATCH (n:Person)
```

```
1 SELECT n.name
2 FROM MATCH (n:Person) FOR TX_TIME AS OF CURRENT_TIMESTAMP
```

The predicate **`AS OF`** `{timestamp}` is used to see the graph as it was at a specific point in time in the presence or past. The following example query retrieves the graph as it was on 1st February 2020 at 1 pm.

```
1 SELECT n.name
2 FROM MATCH (n:Person) FOR TX_TIME AS OF TIMESTAMP '2020-02-01 13:00'
```

The next query using the **`BETWEEN`** `{timestamp}` **`AND`** `{timestamp}` predicate will show all graph elements that were visible at any point between two specified points in time. It works inclusively, i.e., an element visible exactly at the start or exactly at the end will be added to the result set, too.

```
1 SELECT n.name
2 FROM MATCH (n:Person) FOR TX_TIME
3     BETWEEN TIMESTAMP '2020-02-01 12:00' AND TIMESTAMP '2020-02-28 12:00:00'
```

The extension `FROM {timestamp} TO {timestamp}` will also show all elements that were visible at any point between two specified points in time, including start, but excluding end.

```
SELECT n.name
FROM MATCH (n:Person) FOR TX_TIME
    FROM TIMESTAMP '2020-02-01 12:00' TO TIMESTAMP '2020-02-28 12:00:00'
```

To query for the current state and complete history of a given pattern the predicate `ALL` can be used.

```
SELECT n.name
FROM MATCH (n:Person) FOR TX_TIME ALL
```

In PGQL, it is possible to define multiple patterns within a single `FROM` clause by using multiple match clauses separated by a comma. Our extension can be used within each of these match expressions. This provides a flexible mechanism to define patterns with parts occurring at different times. For example, to find people who currently liked a post that already existed on January 1st, 2020, the following query can be used.

```
SELECT m.firstName, m.lastName
FROM MATCH (p:Post) FOR TX_TIME AS OF DATE '2020-01-01',
     MATCH (m:Person)-[:likes]->(p) // current
```

For example, this powerful mechanism could be used in IoT applications to find a sensor that is currently connected to an asset that existed in the past. Besides a path pattern, a graph pattern is a concatenated list of path patterns in round brackets, which can be also specified after the match keyword. If a transaction time predicate should be applied to a set of path patterns, it can be used after such a graph pattern, as can be seen in the following example.

```
SELECT m.firstName, m.lastName
FROM MATCH (
        (p:Post)-[:hasTag]->(t:Tag)-[:inClass]->(tc:TagClass),
        (m:Person)-[:likes]->(p:Post)
    ) FOR TX_TIME AS OF DATE '2020-01-01'
```

Bitemporal Queries

A TPGM$^+$ graph has both a managed transaction- and valid-time domain. Graph elements, as well as their properties, are associated with both transaction-time and valid-time periods. This concept is very useful for use cases where it is necessary to capture both the periods during which facts were believed to be true in real-world as well as periods during which those facts were recorded in the database.

For example, a student changes his address. Typically the address changes legally at a specific time, but it is not changed in the database concurrently with the legal change. In that case, the transaction-time period automatically records when the new address is known to the database, and the valid-time period records when the address was legally effective. Successive updates to bitemporal graphs can journal complex twists and turns in the state of knowledge captured by the database [120].

Queries on bitemporal graphs can specify predicates on both dimensions to qualify rows that will be returned as the query result. For example, the following query returns all

students of Universities in Leipzig who are matriculated as of December 1, 2019, recorded in the graph database as of January 1, 2020.

```
SELECT n.name
FROM MATCH (n:Person)-[s:studiedAt]->(u:University)
    ON student_network FOR TX_TIME AS OF DATE '2020-01-01'
WHERE u.city = 'Leipzig'
    AND s.VAL_TIME CONTAINS DATE '2019-12-01'
```

Query the Evolution of a Property

Every vertex and edge can have zero, one or more properties in form of key-value pairs, where the key represents the name of the property. For every property, transaction-time versioning is supported to track the addition of new properties, changes in values or deletion of exiting properties. Thus a property behaves like a vertex or an edge. By inserting a vertex or edge with properties into the database, each gets the same system-time period as the respective vertex or edge. The previous introduced `FROM` clause extensions have also an effect on the property retrieval since their transaction-time period will be considered, too.

In addition, each property contains a valid-time period. For example, each `University` vertex has a property `studentCount` whose value is periodically updated. Each value has an application time period that defines, for which time the value was true. If no further condition is specified for an application time enabled property, all values are returned without information about the validity. The following query finds all university vertices and outputs the name and the value of the property `studentCount`.

```
SELECT u.name AS name, u.studentCount AS cnt
FROM MATCH (u:University)
WHERE u.city = 'Leipzig'
```

The since the property has three different values for different valid time periods, the resulting table consists of three rows.

name	cnt
Leipzig University	28004
Leipzig University	28797
Leipzig University	29061

To retrieve the information of the validity of the values, the period of validity can be selected, as shown in the next query.

```
SELECT u.name AS name, u.studentCount AS cnt, u.studentCount.VAL_TIME AS val
FROM MATCH (u:University)
WHERE u.city = 'Leipzig'
```

The result of the query is given in the following table. Since the return value of a period identifier is a period, the values are represented as intervals with the according period bounds.

All previously introduced conditions of the `WHERE` clause that are applicable for vertices and edges can be used on properties too. In the following example, we use the `CONTAINS` predicate to get the number of students for this time.

name	cnt	val
Leipzig University	28004	[2016-04-01 00:00, 2017-04-01 00:00)
Leipzig University	28797	[2017-04-01 00:00, 2018-04-01 00:00)
Leipzig University	29061	[2018-04-01 00:00, 2019-04-01 00:00)

```
SELECT u.name, u.studentCount, u.studentCount.VAL_TIME as val
FROM MATCH (u:University)
WHERE u.city = 'Leipzig'
    AND u.studentCount.VAL_TIME CONTAINS TIMESTAMP '2018-01-01 00:00'
```

The resulting table thus consist just of one row, as shown below.

name	cnt	validity
Leipzig University	28797	[2017-04-01 00:00, 2018-04-01 00:00)

6.3.3 Aggregations

An aggregate function allows performing a calculation on a set of values to return a single scalar value. Aggregate functions are used with the `GROUP BY` and `HAVING` clauses of the query.

In this work, the PGQL language was extended by two functions: `FIRST({date or time values})` and `LAST({date or time values})`. The former returns the chronological earliest date or timestamp, while the latter returns the chronological last.

The following query asks for the the first beginning of a *studentAt* relationship according to the application-time and system-time domain.

```
SELECT FIRST(s.VAL_FROM) AS earliestStart,
       FIRST(s.TX_FROM) AS earliestTx
FROM MATCH ()-[s:studiedAt]->()
```

The result of the query is given in the table below, which consists of one row that contains the aggregated values.

earliestStart	earliestTx
1409-04-01 00:00:00	2006-05-12 14:45:22

To bring another example, the following query can be used to answer the question: "For universities of a certain city, when was the first time a student began his studies, and when was the most recent time?"

```
SELECT u.city,
       FIRST(s.VAL_FROM) AS earliestStart,
       LAST(s.VAL_FROM) AS latestStart
FROM MATCH (n:Person)-[s:studiedAt]->(u:University)
GROUP BY u.city ORDER BY earliestStart
```

An exemplary result of this query is given in the following table. All matching subgraphs are grouped by the city property of university nodes. For all grouped paths, the first and most recent start of studies is returned through the aggregation.

city	earliestStart	latestStart
Leipzig	1409-04-01 00:00:00	2020-04-01 00:00:00
Munich	1472-04-01 00:00:00	2020-04-01 00:00:00
Berlin	1810-04-01 00:00:00	2020-04-01 00:00:00

6.4 Continuous Graph Query Notifications

We introduced T-PGQL (Section 6.3) as a query language for executing SELECT queries on TPGM$^+$ (Section 6.2) graphs. Querying a graph with T-PGQL is usually done in a single graph query execution called *one-time query* [22], i.e., a user formulates a SELECT query, executes that query on the current state (an isolated snapshot) of a database system that maintains a TPGM$^+$ graph and gets a result in form of a table back. In this way, one can query for current and historical data of the graph where one query leads to one fixed result. Subsequent changes of the graph are not taken into account unless the query is executed again which leads to a result that recognizes all transactions that are made until the time when the query was executed.

Talking about changes in a graph leads to the observation of events. An *event* represents an occurrence of interest at a point in time [6], e.g., an asset's sensor captures a temperature or a user liked a post in a social network. We semantically distinguish between *application-world events* and *transaction-world events*. Former is an event that happened in the observed real world. In the graph context, such an event can be described by a graph pattern and its predicates inside a graph query. In contrast, the discovery of an instance of a pattern in the data store at a specific time is a *transaction-world event*, i.e., the most recent commit (1) created an instance of a pattern that did not exist before, e.g., a captured temperature of a sensor exceeds a threshold, or (2) destroys an instance of the pattern that already existed, e.g. a friendship relation between two users of a social network is removed.

The question arises, how a user can get a notification about a *transaction-world events*, i.e., when graph data changes that affect the elements that are accessed to create the query result. For RDBMS, there is a feature called *Continuous Query Notification (CQN)* [149], which is currently implemented by the Oracle database. It allows to register a SQL query and receive notifications when an event occurs that changes a table, i.e., that rows have been updated. This is useful in applications like near real-time monitoring, auditing applications, or for such purposes as mid-tier cache invalidation [149, 201].

In the collaborative project that this section is about, approaches were developed to apply these concepts of the relational CQN to graph databases. In the following only conceptual approaches are described. A reference implementation and evaluation is part of future work. For now, only a bachelor thesis [227] has implemented parts of it.

A requirement of a graph database that implements the TPGM$^+$ is to not only execute a T-PGQL query, but also to register one as a *continuous query* [201] that notifies a recipient if the graph changes in a way that affects the query or its result. We call this a *Continuous Graph Notification (CGN)*. A registration should be configured by the T-PGQL SELECT query itself, a registration validity period that specifies when the notification is enabled and when it will be disabled, a notification endpoint (e.g., a messaging queue) and a notification type. For the latter, there are two types of graph query notifications:

1. the *Graph Change Notification (GCN)* which creates a notification, if an event that changed the queried graph *affects the graph elements queried* (what does not automatically mean that the query result has also changed); and
2. the *Graph Query Result Change Notification (GRCN)* which creates a notification, if an event that changed the queried graph in a way that *affects the query result.*

Note that both types are a evaluation of a registered query and no query re-execution. Thus, the recipients are notified about the event but do not get the updated query result.

Assume the following example T-PGQL query. The query describes the event of a temperature measurement above a value of 40 from a sensor that is part of an asset that is connected to an asset with id '42'. The projection of the query is the sensor value and its validity timestamp, which describes the time when the measure happened in the observed real world.

```
SELECT s.value, s.value.VAL_FROM
FROM MATCH (a1:Asset)-[p:partOf]->(a2:Asset)-[:hasSensor]->(s:Sensor)
WHERE a1.id = 42 AND s.type = 'temperature' AND s.value > 40
```

We assume that there are several matches for this pattern without recognizing the predicate of the value threshold, but none that fulfills this condition (i.e., all temperature values are below a value of 40). Further assume, that at a time ω_1, the value of a sensor's property is updated from 39 to 40. If the query is registered by a GCN, a notification about that event is created, since one of the involved graph elements changed its state, but the query result is not affected. If the query is registered by a GRCN, no notification is created, since the query result does not change (it is still empty). Now, assume that at time ω_2 the value of the sensor changes from 40 to 41. At this time, a notification is created for both types, since 1) a graph element that is part of the pattern changes and 2) the query result changes in the form that now one row is part of the result.

6.5 Conclusion

This section summarises the results of a one-year research collaboration between Oracle Labs and the University of Leipzig. The goal of the collaboration was to show that temporal graphs are suitable for modelling changing relationships between entities that produce time series data themselves. This organising principle was illustrated by three contributions. First, we presented a bitemporal graph data model TPGM$^+$, which, unlike its predecessor TPGM, also supports the evolution of property values. We also presented a property graph query language, T-PGQL, which adds temporal extensions to Oracle's existing PGQL language. The syntax as well as the usability was demonstrated by means of various examples. Finally, an approach to implementing continuous queries using such T-PGQL queries was presented. These so-called CGNs react to events that change the graph and then generate a notification when relevant parts of the graph or the corresponding query result change. Due to the short duration of the project, formal definitions, evaluations and prototypical implementations are part of future work.

It is worth mentioning that this work inspired a recent research project called *HyGraph* [43], funded by the German DFG and French ANR starting in 2023 for a period of 3 years.

7

Seraph: Continuous Queries on Property Graph Streams

The versatility and expressive power of graphs have prompted the development of various graph models and query languages utilized by practitioners to simulate real-world phenomena. The property graph model, along with its associated query languages - one of which is the prevalent Cypher- have gained widespread use both in industry and academia. Real-time data analysis and management are increasingly critical for modern businesses, however, graph query languages lack the necessary features to handle streaming graph data and continuous query evaluation.

This work proposes Seraph, a Cypher-based language that supports native streaming features within industry-ready property graph query languages. We formally define the Seraph semantics by combining stream processing with property graphs and time-varying relations, propose its syntax, and demonstrate the usage of Seraph for emerging graph-based continuous queries in real-world industrial use cases.

The contents of this section are published under the title *Seraph: Continuous Queries on Property Graph Streams* [180].

7.1 Introduction

With the growing availability of information, interconnected data have become pervasive. Graphs, in particular, Property Graphs (PGs) [14] (also denoted as Labeled Property Graphs (LPGs)), are a widespread data model in many industrial domains such as healthcare, social media, cybersecurity, fraud detection, and genomics. Coherently, declarative graph query languages like Cypher [71], G-CORE [16], and PGQL [166] have emerged as the formalisms of choice for expressing sophisticated information needs declaratively. Moreover, the efforts above are converging into GQL [55, 98], the future standard graph query language that will pave the road to workload portability and shared consensus to manipulate PGs.

Graphs not only exhibit a notable increase in volume but also demonstrate a significant level of dynamism [29]. When slow in frequency, the changes on graphs can be addressed by temporal graph data models and corresponding query languages [176] that include

operators for exploring graph versions across time. On the other hand, a paradigm shift in the query model is needed when the frequency of changes grows and directly impacts the high throughput and low latency of query results, as it often occurs in streaming graph settings [155].

Continuous queries (CQs) [19, 201] are a class of queries that repeatedly reports the results until explicitly terminated. CQs are typically evaluated over data streams, i.e., unbounded sequences of timestamped data items [23, 201]. To deal with the unboundedness of the input streams, CQs include operators that leverage data timestamps and operate the query evaluation by recency.

In practice, CQs enable reactive analytics, and thus, they are popular in stream processing domains like network monitoring, real-time surveillance, micro-mobility. In such domains, it is of paramount importance to output the results of the query before the data becomes stale. However, the high cost of designing and maintaining custom stream processing pipelines has paved the road to declarative continuous query languages [85]. The database literature shows that the declarative paradigm poses significant advantages in such domains, e.g., interoperability across systems, optimisation opportunity, and simplicity of use [154].

Now that CQs are becoming relevant for various property-graph-centric task [155, 189], it is important to bridge the gap for a declarative PG continuous query language. Table 7.1 shows examples of CQs for the stream processing domain mentioned above that would benefit from a graph stream data model: the first query, which devotes to **network monitoring**, asks for paths that denote anomalies in the routes to the egress switch (i.e. a router responsible for outgoing traffic in a time-based interval); the second query, which devotes to **real-time surveillance**, asks for computing the list of persons who passed by a crime scene within 30 minutes; the last query, which relates to **micro mobility**, looks for the violations of a business rule for limited time free bike rides in a given temporal lapse. It is worth noting here that the above CQs not only need to be repeatedly evaluated for incoming data, but they must also identify a temporal pattern to bind the results.

On the one hand, the CQs presented above show the need for enriching current declarative graph query languages with the necessary abstractions and expressive power to formulate continuous graph queries. However, current popular graph query languages, such as Cypher, lack these abstractions. Moreover, algebraic frameworks for streaming graph queries started to be defined, along with data and query models and preliminary optimizations [155] but as of today, these cannot be encoded as bulk queries in a declarative

Domain	**Examples of Graph Continuous Queries**
Network Monitoring	What are the anomalous routes that connect to the egress switch **during the last 15 minutes**?
Real-Time Surveillance	Who has passed by a given crime scene **in the last 30 minutes**?
Micro Mobility	Did anyone violate the 20-minute free renting limit **in the last hour**?

Table 7.1: Summary of continuous information needs for use-cases in three different domains.

language. On the other hand, existing continuous declarative languages are either limited to the relational model [210] or limited to RDF. The former are incapable of expressive complex analytics such as path queries. At the same time, the latter have focused on the relational subset of SPARQL [209], neglecting advanced features such as regular path queries [155].

In this work, we fill this gap and focus on the language aspects of graph continuous queries. Precisely, we present the design of syntactic and semantic components of a Cypher-based continuous query language for streaming property graphs, namely **Seraph**. Motivated by other ongoing GQL standardization efforts around graph query and schema languages [18, 55, 98], in which formal semantics need to be defined prior to any implementation, we define the syntax and semantics of **Seraph** and properly formalize the latter to avoid underlying ambiguities and incorrect behavior of the queries.

In designing such a language, we elicit a set of design requirements acquired from our industrial-strength use cases:

R1 **Declarative semantics.** The language must be declarative to guarantee interoperable execution across systems, optimizations, and simplicity of adoption.
R2 **Continuous evaluation.** The language must have operators that allow repeated evaluation over time, i.e., choosing a time interval and a sequence to evaluate the query.
R3 **Result emitting.** The language must include operators that allow controlling the report of results, i.e., what is part of the result and when it will be ready to be emitted.
R4 **Preserving expressiveness.** The language should preserve the expressiveness of the base language for querying a PG, i.e., everything that can be expressed in the base language can be expressed in the extended language.

Seraph results from a long-term collaboration between several academic institutions and Neo4j. It conjugates Cypher, a widely used graph query language with a key role in the ongoing GQL standardization, with windowing mechanisms at the core of continuous queries over streams [19, 32]. While the very first version of GQL (without temporal extensions) is expected in 2024, at the moment GQL is only available to the members of the standardization committee (ISO/IEC JTC1 SC32 WG3 Database Languages). It will take some time for GQL to be implemented into products even after the publication of the standard. This justifies and motivates our choice of focusing on Cypher as a basis for Seraph, given the wide availability of the former in several industrial products/use cases and its closeness to GQL. We believe that our work will help reach a consensus for the future temporal expansions of GQL, which are certainly deemed important but not included in the first version.

As a result, Seraph stands as a productive and industry-ready language that allows querying graph streams. The proposed language is simple, intuitive and highly expressive at the same time. Anyone who is familiar with Cypher and can address a problem in a static property graph setting will be able to use Seraph in a streaming context. The language and its detailed formalization, as presented in our work, lay the foundations for future implementations and are the necessary steps to be carried out in order to guarantee their underlying correctness.

In summary, we make the following main contributions, which address the challenges C1, C3 and C10 from Section 1.2:

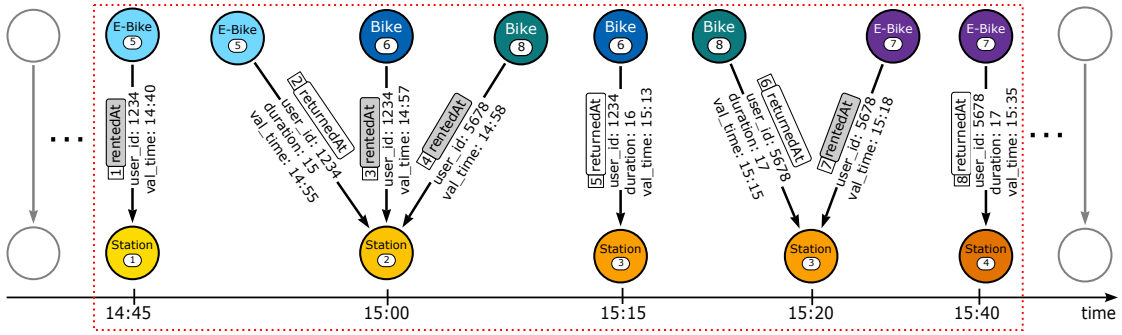


Figure 7.1: Stream of property graphs representing the events captured into the RideAnywhere Kafka queue.

- we present the data model underlying graph continuous queries and formally describe the duality between a stream of property graphs and time-varying relations;
- we lay the foundations of a Cypher-based query model for continuous queries over property graphs streams by using the concept of snapshot reducibility from temporal relational databases;
- we formally define the syntax and semantics of Seraph, an easy-to-use Cypher-based query language that incorporate primitives for continuous evaluation. The latter is, to the best of our knowledge, nonexistent in current graph query languages. In contrast, they are urgently needed in industry-wide applications and desirable for the development of ongoing GQL standards.

The remainder of the section is organised as follows: We will go into details of the micro-mobility example in Section 7.2. We provide an overview of the core of Cypher in Section 7.3, giving the preliminary knowledge needed to formalize Seraph, and explaining how a Cypher-only solution won't satisfy the requirements. Section 7.4 describes further two industrial use-cases and the use of Seraph to answer the continuous questions in Table 7.1. Section 7.5 contains the formal specification of the semantics and syntax of Seraph together with a solution of the running example. In Section 7.6, we give an outlook on future implementations, while Section 7.7 concludes the section. For related work on graph streams and graph stream languages, we refer to Chapter 2.

7.2 Running Example

In this section, we describe the use case of fraud detection in the micro mobility domain mentioned in the introduction. The scenario of a fictional vehicle sharing provider described below is inspired by a real business scenario within a company namely *nextbike BY TIER* (`https://www.nextbike.de`) from Leipzig, Germany. This company applies graph technologies for demand prediction, usage increase and optimization of rental stations and zones and their locations.

The company *RideAnywhere* is known as a leading company that operates public bike, scooter and car-sharing systems. Various rental stations and zones are available within a city, which offer electrically operated cars, bicycles (e-bikes) and e-scooters, and classic bicycles. A user can use a mobile app to rent an available vehicle at a rental station. RideAnywhere offers various price models: from half-hourly to monthly subscriptions. If

a vehicle is returned, the app calculates the total price of the rental based on the duration and the existing subscription.

Each rental station is connected to RideAnywhere's headquarters through a 4G connection. Rental and return events are transmitted to a central Kafka event queue, and these transmissions occur every 5 minutes for energy and traffic preservation. A collective event logs all of the rentals and returns of the preceding 5 minutes and includes the vehicle ID, the station ID, and user information. We appropriately model the relationship within the event as a property graph, composed of station and vehicle nodes and relationships that represent rentals and returns. Additionally, we provide properties indicating rental time, return time, duration, and unique identifiers for vehicles, users, and stations.

The timeline reported in Figure 7.1 illustrates the events arriving in the Kafka queue of the RideAnywhere headquarter from 14:45h to 15:40h on a day in August 2022. Each event represents a property graph that contains rentals of a 5 minute period. Let us discuss the events of this example in detail.

14:45h An E-bike with id 5 was rented at station with id 1. The rental was done at 14:40h from a user with id 1234, which is stored in the edge properties. No further rentals or returns happened in the period [14:40,14:45).

15:00h The E-bike with id 5 was returned at station 2 at 14:55h. At the same station, two other bikes were rented. One of them from the same user, the other of user with id 5678.

15:15h The bike with id 6 was returned at station 3 at 15:13h.

15:20h Again at station 3, bike 8 was returned by user with id 5678 at 15:15h and an e-bike with id 7 was rented again by the same user 3 minutes later.

15:40h The E-bike with id 7 was returned at station 4 at 15:35h. No further rentals or returns happened in the period [15:35, 15:40).

Using the Neo4j Kafka Connector [141], all incoming events are merged and persisted in a Neo4j graph database. Vertices sharing the same identifier (e.g., for stations and bikes) will be merged to a single vertex. For the example graph stream of Figure 7.1, the resulting merged property graph of the interval from 14:45h to 15:40h is visualized in Figure 7.2. The graph consists of four station and four bike nodes as well as four rentals of two users represented by eight timestamped relationships. A closer look reveals a pattern for both users, which will be discussed in more detail shortly.

To make the service attractive and affordable for students, an additional pricing model was developed with the local student union a few months ago. It provides that the first 20 minutes of e-bike or bicycle rental are free for valid students. If a vehicle is returned by a student and the 20-minute rental period has not been exceeded, no fee will be charged. If the time period is exceeded, the regular rate will be charged. RideAnywhere's goal is to increase the number of younger customers, achieve broader use of the service and generate more revenue by exceeding free times.

Shortly after its introduction, the student offer was widely used. The number of rentals increased by 35% compared to the average monthly usage before the student offer. However, after the first 3 months with this model, an analysis on the rental-graph found that students rarely made rentals longer than 20 minutes: only 5% of all student rentals per month. The RideAnywhere analytics team suspects that longer distances are covered by renting a vehicle again shortly thereafter (at a 5 minute interval), which is prohibited

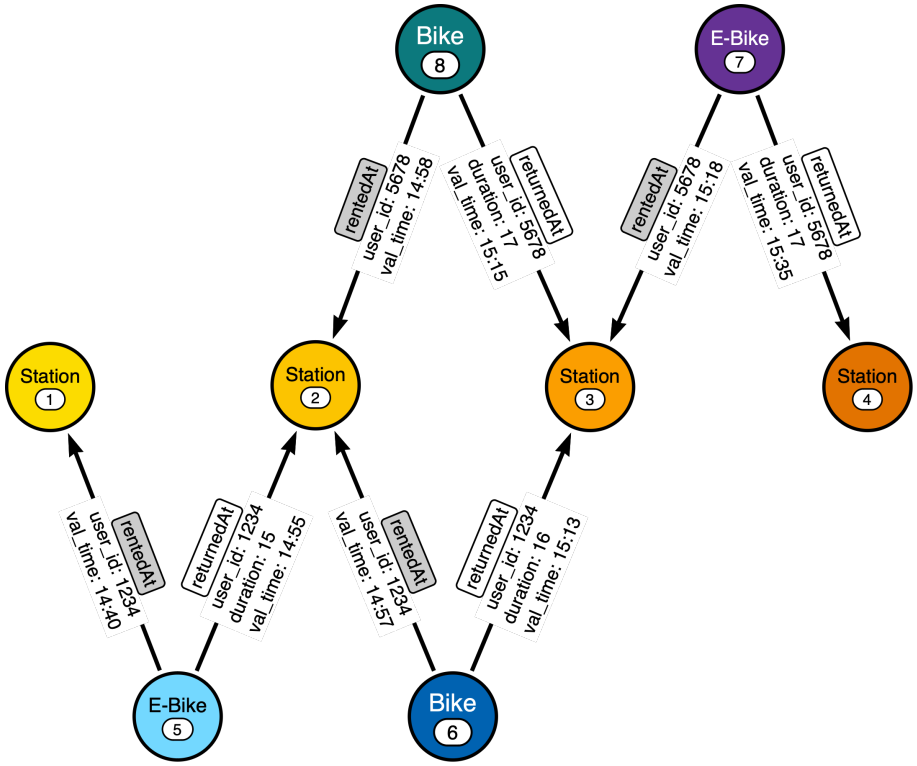


Figure 7.2: Graph resulting from loading the events from 14:45h to 15:45h into a Neo4j graph database.

by company policy. This trick allows students to cover any length of distance using the free period multiple times.

The data analytics team is now expected to find a way to continuously detect users who use this trick so that they can be immediately alerted that this will lead to expulsion from the student offer by repeated violation. We present two solutions to this problem: one with Cypher including its drawbacks in Section 7.3.3, and one using Seraph that shows its strength in Section 7.5.4. It should be noted that the selected example with only minute-by-minute data does not do justice to the real-time character of the possibilities of continuous querying on a graph stream, but is suitable for demonstration.

7.3 Background: the Cypher language

In the following, we provide as the necessary background the formal specification of a core subset of Cypher [71] for static property graphs. It consists of a data model including values, graphs, and tables (Section 7.3.1) and the query language along with its evaluation semantics including expressions, patterns, clauses, and queries (Section 7.3.2). Finally, we discuss the drawbacks of Cypher-only solution to the running example from Section 7.2 (Section 7.3.3).

7.3.1 Data model

Although a property graph has already been defined in Section 2.1, a property graph is defined below according to Francis et al. [71], as Seraph is based on Cypher and this is

formally defined on this definition.

Cypher's data model is based on three first class objects, namely values, property graphs, and relations, the latter being referred to as tables in Cypher's terminology. From [71], we consider three disjoint sets $\mathcal{K}$ of property keys, $\mathcal{N}$ of node identifiers, and $\mathcal{R}$ of relationship identifiers. These sets are all assumed to be countably infinite. The set $\mathcal{V}$ of values contains multiple inductively defined elements. We assume two base types: the integers $\mathbb{Z}$, and the type of finite strings over a finite alphabet Σ.

Definition 25 (Property graph [71]). *Let $\mathcal{L}$ and $\mathcal{Y}$ be countable sets of node labels and relationship types, respectively. A property graph is a tuple $G = (N, R, src, trg, \iota, \lambda, \kappa)$ where:*

- *N is a finite subset of $\mathcal{N}$, whose elements are referred to as the nodes (also denoted as vertices) of G.*
- *R is a finite subset of $\mathcal{R}$, whose elements are referred to as the relationships (or edges) of G.*
- *src and trg are functions $R \to N$ that map a relationship to its source and target node, respectively.*
- *$\iota : (N \cup R) \times \mathcal{K} \to \mathcal{V}$ is a finite partial function that maps a pair (node|relationship,property key) to a value.*
- *$\lambda : N \to 2^{\mathcal{L}}$ is a function that maps each node id to a finite (possibly empty) set of labels.*
- *κ: $R \to \mathcal{Y}$ is a function that maps each relationship identifier to a relationship type.*

For example, the graph of Figure 7.2 is formally represented in this model as a graph $G = (N, R, src, trg, \iota, \lambda, \kappa)$:

- $N = \{n_1, \ldots, n_8\}$; $R = \{r_1, \ldots, r_8\}$;
- $src = \{r_1 \mapsto n_5, r_2 \mapsto n_5, r_3 \mapsto n_6, r_4 \mapsto n_8, \ldots\}$;
- $trg = \{r_1 \mapsto n_1, r_2 \mapsto n_2, r_3 \mapsto n_2, r_4 \mapsto n_2, \ldots\}$;
- $\iota(r_1, \text{user\_id}) = 1234, \iota(r_1, \text{val\_time}) = 14{:}40, \ldots$;
- $\lambda(n_1) = \lambda(n_2) = \lambda(n_3) = \lambda(n_4) = \{\text{Station}\}$,
  $\lambda(n_5) = \lambda(n_7) = \{\text{E-Bike}\}, \lambda(n_6) = \lambda(n_8) = \{\text{Bike}\}$;
- $\kappa(r) = \begin{cases} \text{rentedAt} & \text{for } r \in \{r_1, r_3, r_4, r_7\}, \\ \text{returnedAt} & \text{for } r \in \{r_2, r_5, r_6, r_8\}. \end{cases}$

The flexibility of the property graph model also allows the modelling of hierarchies, e.g. by using multiple type labels per node (see λ), e.g., `:superclass:subclass` or dedicated relationship types, e.g., `(a)-[:isSubclassOf]->(b)`.

Definition 26 (Tables [71]). *Let $\mathcal{A}$ be a countable set of names. A record is a partial function from names to values, conventionally denoted as a tuple with named fields $u = (a_1 : v_1, \ldots, a_n : v_n)$ where $a_1, \ldots, a_n$ are distinct names, and $v_1, \ldots, v_n$ are values. The order in which the fields appear is only for notation purposes. We refer to $dom(u)$, i.e., the domain of u , as the set $\{a_1, \ldots, a_n\}$ of names used in u . We use $()$ to denote the empty record, i.e., the partial function from names to values whose domain is empty.*

If $\mathcal{A}$ is a set of names, then a table with fields $\mathcal{A}$ is a bag, or multiset, of records u such that $dom(u) = \mathcal{A}$. A table with no fields is just a bag of copies of the empty record. Lastly, we define the bag difference of two tables T and T' as their bag difference, i.e., $T \setminus T'$.

7.3.2 Query language

The Cypher query language, whose syntax is presented in Figure 7.3, includes expressions, patterns, clauses, and queries. Due to the limited space, like in [71] we only focus on the latter two. A query is either a sequence of clauses ending with the **RETURN** statement, or a union of two queries. The semantics of queries associates a query Q and a graph G with a function $[[Q]]_G$ that takes a table and returns a table. Notably, the semantics of a query Q is a function and should not be confused with the output of Q. The evaluation of a query starts with the table containing one empty tuple, which is then progressively changed by applying functions that provide the semantics of Q's clauses. The composition of such functions, i.e., the semantics of Q, is a function again, which defines the output as:

$$output(Q, G) = [[Q]]_G(T())$$

where $T()$ is the table containing the single empty tuple ().

Let us have a look at the semantics of a pattern. The **MATCH** clause extends the set of field names of T by adding field names that correspond to names occurring in the pattern but not in u (the value to field assignments). It also adds tuples to T, based on found matches of the pattern in graphs. We show how to compute $[\![\textbf{MATCH}\ \pi]\!]_G(T)$, where π is the path pattern [71] to search for.

$$[\![\textbf{MATCH}\ \pi]\!]_G(T) = \biguplus_{u \in T} \{u \cdot u' | u' \in match(\pi, G, u)\}$$

The pattern π is evaluated on the graph G and extending T by adding field names and tuples based on the matches found in G. Each existing assignment $u \in T$ is extended by the assignments u' that are part of the finite set $match(\pi, G, u)$, which gives the semantics of the pattern matching of Cypher. Note that a pattern with variable length can be subsumed by a (possibly infinite) set of fixed length patterns, so-called rigid patterns [71]. Let π be a path pattern, $free(\pi)$ the union of all free variables of each node and relationship pattern occurring in π, $rigid(\pi)$ the set of all rigid patterns subsumed by π, G the graph, u an assignment, $dom(u)$ the domain of u (set of names) and p a path with node ids from N and relationship ids from R, the set is defined as:

$$match(\pi, G, u) = \biguplus_{\substack{p \in G \\ \pi' \in rigid(\pi)}} \left\{ u' \;\middle|\; \begin{array}{c} dom(u') = free(\pi) - dom(u) \\ \wedge (p, G, u \cdot u') \vDash \pi' \end{array} \right\}$$

Note that $\biguplus$ is a bag-union, i.e., a new occurrence u' is added to $match(\pi, G, u)$ if a new combination of π' and p is found that the pattern matching relation holds: $(p, G, u \cdot u') \vDash \pi'$,

```
query  ::= query° | query UNION query | query UNION ALL
query° ::= RETURN ret | clause query°
ret    ::= * | expr [AS a] | | ret , expr [AS a]
clause ::= [OPTIONAL] MATCH pattern_tuple [WHERE expr] | WITH ret [WHERE expr
    ] | UNWIND expr AS a
pattern_tuple ::= pattern | pattern , pattern_tuple
```

Figure 7.3: Syntax of queries and clauses of Cypher [71].

r.user_id	s.id	r.val_time	hops
1234	1	14:40	[2,3]
5678	2	14:58	[3,4]

Table 7.2: Results of the Cypher query in Listing 7.1 at 15:40h.

i.e., a path p in a graph G satisfies a pattern π under the assignments $u \cdot u'$ of values to the free variables of the pattern. Additional details of the Cypher semantics can be found in Francis et al. [71].

7.3.3 Running Example vs Cypher

Following up on the Section 7.2 example to detect subsequent rentals of the same user in the last hour, we designed the Cypher query shown in Listing 7.1 that implements one possible but limited solution representing this pattern. The drawbacks of this solution are evaluated at the end of this section.

The first part from line 1 to line 2 defines two timestamps as bounds of a 1h window from the moment of the query execution. Lines 3-4 define the patterns: A bike was rented at a station *s* from which a path with at least a length of 3 relationships ends at a station *o*. The dynamic recursive pattern is assigned to a path variable *q* for later use. Lines 7-8 define a condition that the timestamp of all relationships of the path *q* have to be in the 1h window. The selection at lines 9-10 ensure the same user for all rentals and returns, guarantee that the first rental ended chronologically before the second starts (line 9) and both rentals do not exceed the free period of 20 minutes (line 10). The user id, time of the first rental and ids of all involved stations (derived in line 6) are returned, as stated in line 11.

Table 7.2 reports the result of the query evaluation at 15:40, showing that in the last hour, the users with ids 1234 and 5678 illegally extended their free rental time each by a second subsequent rental.

However, this one-time Cypher query computes the information need for the graph changes of one specific hourly interval, but has several drawbacks. First, the PG data model on which Cypher is based is static, i.e., there is no support of a continuous stream

```
1  WITH datetime() - duration('PT60M') AS win_start,
2       datetime() AS win_end,
3  MATCH (:Bike)-[r:rentedAt]->(s:Station),
4        q = (b)-[:returnedAt|rentedAt*3..]-(o:Station)
5  WITH r, s, q, relationships(q) AS rels,
6      [n IN nodes(q) WHERE 'Station' IN labels(n) | n.id] AS hops
7  WHERE ALL(e IN rels WHERE
8      win_start <= e.val_time <= win_end AND
9      e.user_id = r.user_id AND e.val_time > r.val_time AND
10     (e.duration IS NULL OR e.duration < 20) )
11 RETURN r.user_id, s.id, r.val_time, hops
```

Listing 7.1: Cypher query to retrieve users using the free period for two subsequent rentals in the last hour.

of graph elements. Further, the query has to be continuously evaluated on the most recent events and the results need to be computed every 5 minutes from a defined time instant, which results in the continuous evaluation requirement **R2**. Moreover, they want to get user 1234 returned at 15:15 and user 5678 at 15:40, thus, only new results as soon as the last event arrived in the last hourly period, which results in the result emitting requirement **R3**.

With Cypher, this could be realized only by external code that executes this query every 5 minutes. However, such a workaround would break the declarative paradigm (violating **R1**). Moreover, the underlying system would be unaware of the continuous semantics, which would almost certainly lead to suboptimal query evaluation and possibly incorrect execution. In fact, each query will run isolated from the other, possibly considering caching mechanism design for the static case. Thus, a language like Cypher lacks a query mechanism that natively controls the continuous evaluation of the query and the result emission while at the same time preserving the expressive power of the non-streaming language, which leads to the expressiveness requirement **R4**.

7.4 Seraph By Examples

Before we get into the technical definitions, we pick up the two industrial use cases from Section 7.1 to justify the design (w.r.t. the requirements) and formalization of Seraph. The goal is to give high-level intuition of how a Seraph query looks and clarify what queries we target in this work. To simplify understanding of the syntax extensions, the original Cypher keywords are shown in blue and those introduced by Seraph are shown in green.

7.4.1 Network Monitoring

Computer networks span all levels of the stack, from physical connections up to mobile and microservices constituting a company's cloud. Graphs offer a natural way of modelling such scenarios and performing network optimization, asset management and inventory mapping. Network management is thus intrinsically a graph problem. While graph query languages like Cypher play a key role in investigating dependencies and in running diagnostic analyses (e.g., the root cause of a past network fault), Seraph offers the possibility to execute network impact analysis continuously.

```
REGISTER QUERY anomalous_routes STARTING AT datetime() {
  MATCH path = allShortestPaths(
  (rack:Rack)-[:HOLDS|ROUTES|CONNECTS*]-(r:Router:Egress))
  WITHIN PT10M
  WITH rack, avg(length(path)) as 10minAvg, path
  WHERE (10minAvg - 5 / 0.5) >= 3
  EMIT path
  SNAPSHOT
  EVERY PT1M
}
```

Listing 7.2: Monitoring computer networks using Seraph.

Let's assume that we model the network endpoints (e.g., servers, routers, switches and racks) of the data center as nodes and the "cables" between them as relationships. For instance, a rack HOLDS a switch that ROUTES an interface that CONNECTS a router in a network. We consider connections redundant if one of the cables gets loose or cut, i.e., the ROUTES relationship between a switch's interface and the network breaks, the number of hops can increase, but no rack can become unreachable. We know from the configuration of the network that the shortest routes from all racks to the egress router require on average 5 hops, but network events may cause this path to be longer, and we observed a standard deviation of 0.3 hops. We can identify anomalous routes using the z-score, i.e., the number of standard deviations σ by which an individual x is above or below the mean value δ of the population with $(x - \delta)/\sigma$. Our patterns are routes whose length has a z-score larger than 3, i.e., it is longer than 99,9% of the paths.

Listing 7.2 illustrates how to encode this need in a Seraph query. At each time instant, an arriving property graph represents the configuration of the entire network. The query uses the **WITHIN** (line 4) and the reporting **EVERY** (line 9) clauses to define a 10 minutes wide sliding window that reports every minute (i.e., PT1M) starting from the current system time (line 1), which meets requirement **R2**. The query finds the shortest paths from each rack to the egress router (line 2 and 3) and computes the average length of those paths in the last 10 minutes (line 5).

If the z-score of those paths related to each rack is greater than 3 (line 6), all paths are emitted for every evaluation, which meets requirement **R3**. Two extensions achieve this: First, using the **EMIT** clause (line 7) to specify the projections for the result stream (here all shortest paths by the path variable) and second by the **SNAPSHOT** streaming operator (line 8), which specifies that for each evaluation all result tuples will be emitted regardless of whether they have already been emitted in the previous evaluation. The result of this continuous query is a stream of so-called time-varying tables containing possibly anomalous routes.

7.4.2 Crime Investigations

From fraud detection to security, encompassing surveillance and contact tracing, investigations often require *connecting the dots*. Data models like POLE (Person-Object-Location-Events) underpin a number of analyses that require the identification of patterns [207]. POLE was originally intended for historical analyses that one can perform using graph query languages like Cypher. However, POLE includes temporal metadata that Seraph can exploit. Thus, it already unlocking a number of additional analyses, including, but not limited to, real-time surveillance and contact tracing.

As we adopt the POLE model for surveillance, we model *crimes* and *calls* as Events, which OCCURRED_AT a Location. Moreover, we assume that a number of smart cameras, which can identify each Person passing by (NEAR_TO), are deployed in different Locations within the city of London. We also consider suspects whoever has been convicted (PARTY_TO) for a crime of the same type as the one reported. Moreover, assuming that on average a person walks about 5km in an hour, we restrict the scope of the monitoring to an area of 3km from the crime scenes and a time range of 15 minutes.

Listing 7.3 illustrates how to encode the information-need above in a Seraph query. The query focuses on the last 15 minutes, reporting every 5 minutes starting at the

```
1  REGISTER QUERY watch_for_suspects STARTING AT datetime() {
2     MATCH (call:Event)-[:OCCURRED_AT]->(l:Location)
3     WITHIN PT15M
4     WITH call, point(l) AS crime_scene
5     MATCH (crime:Event)<-[:PARTY_TO]-(person:Suspect)-[:NEAR_TO]->(last_seen:
      Location)
6     WITHIN PT15M
7     WITH call, crime, person, last_seen, distance(point(last_seen),
      crime_scene) AS distance
8     WHERE distance < 3000 AND call.type=crime.type
9     EMIT person, last_seen, call.description
10    SNAPSHOT
11    EVERY PT5M
12 }
```

Listing 7.3: Looking for suspects in crime scenes using Seraph.

current system time. To this extent, it uses the `WITHIN` clause once per `MATCH` (line 3 and line 6), and controls the results reporting using the `EVERY` clause (line 11) and `STARTING AT` (line 1), which satisfies requirement **R2**. The query monitors the streams of crime reports (Lines 2-4) and crosschecks if anyone, who is identified by a smart-camera, was a convicted criminal (Lines 5-7). To restrict the search space, the query looks only for cameras within 3km from the crime scenes and to those suspects that had taken part in a crime of the same type before. The functions `point()` and `distance()` are user-defined functions to perform geo-spatial comparisons. As an output, the query emits the last seen location, the suspect description, and the crime references by `EMIT` (line 9) and `SNAPSHOT` (line 10), satisfying requirement **R3**. With the queries for both use cases, one can see that Seraph expands Cypher and thus does not reduce the expressiveness, which addresses requirement **R4**.

7.5 Formalization of Seraph

This section presents how Seraph supports streaming computations while preserving the expressiveness of the Cypher language. Throughout all following descriptions, we use the notation summarised in Table 7.3. The three key elements of Seraph are aligned with the requirements of Section 7.1: 1) a **data model** that extends the PG model used by Cypher to model streams of property graphs; 2) a **query model** for continuous query evaluation with full control of reporting (**R2,R3**); and 3) **syntax and semantics** of novel time-aware operators over the aforementioned data model (**R1,R2,R3,R4**).

7.5.1 Data Model

We first explain how the data model of Cypher can be extended to deal with property graph streams, where each graph maintains its evolution. The first component of Seraph's data model is a linearly ordered discrete *time domain* Ω like in [19, 139].

Concept	Notation	Set notation
Time instant / time domain	ω	Ω
Time interval	τ	-
Table	T	-
Time-annotated Table	$\widetilde{T}$	$\widetilde{\mathcal{T}}$
Time-varying Table	Ψ	-
Mapping	u	-
Time-annotated Mapping	μ	-
Property Graph	G	-
Snapshot Graph	$\widetilde{G}$	-
Property Graph Stream	S	-
Property Graph Substream	$\widetilde{S}$	$\widetilde{\mathcal{S}}$
Window	w	W

Table 7.3: Summary of notation conventions.

Definition 27 (Time). *Ω is a infinite sequence of time instants $(\omega_1, \omega_2, \dots) \in \Omega$. A time unit is the difference between two consecutive time instants $(\omega_{i+1} - \omega_i)$ and it is constant. A time interval $\tau = [\omega_o, \omega_c)$ is a left-close right-open interval which starts at ω_o and ends at ω_c. Formally, it holds $\tau = \{\omega_i | \omega_i \in \Omega \wedge \omega_o \leq \omega_i < \omega_c\}$.*

According to the definition above, we can define property graphs arriving in a sequence of time instants as a property graph stream (see Figure 7.1 for an example).

Definition 28 (Property Graph Stream). *A Property Graph Stream S is an unbounded ordered sequence of pairs (G,ω), where:*

- *G is a property graph as per Definition 25, and*
- *ω is a non-decreasing timestamp.*

$$S = ((G_1, \omega_1), (G_2, \omega_2), (G_3, \omega_3), (G_4, \omega_4), \dots)$$

Handling stream unboundedness is essential to Seraph's semantics. Thus, we introduce the notion of a *snapshot graph* that, in turns, builds on the concepts of *property graph substream* and *union of property graphs.*

Definition 29 (Property Graph Substream). *Given a property graph stream S and a time interval $\tau = [\omega_o, \omega_c)$, we denote a finite subset of S in τ as a property graph substream:*

$$\widetilde{S}_\tau = \widetilde{S}^{\omega_o}_{\omega_c} = \{(G, \omega) | (G, \omega) \in S \wedge \omega \in \tau, \Omega \wedge \omega_o \leq \omega < \omega_c\}$$

Definition 30 (Union of two Property Graphs). *Assume that $G_1 = (N_1, R_1, src_1, trg_1, \iota_1, \lambda_1, \kappa_1)$ and $G_2 = (N_2, R_2, src_2, trg_2, \iota_2, \lambda_2, \kappa_2)$ are Property Graphs. Under unique name assumption (UNA) [199], we define the union of two Property Graphs as:*

$$G_1 \cup G_2 = \begin{pmatrix} N_1 \cup N_2, R_1 \cup R_2, src_1 \cup src_2, trg_1 \cup trg_2, \\ \iota_1 \cup \iota_2, \lambda_1 \cup \lambda_2, \kappa_1 \cup \kappa_2 \end{pmatrix}$$

, if both graphs are consistent otherwise $G_1 \cup G_2 = \varnothing$.

G_1 and G_2 are consistent iff $\forall r \in R_1 \cap R_2$, it holds that $src_1(r) = src_2(r)$, $trg_1(r) = trg_2(r)$, $\kappa_1(r) = \kappa_2(r)$ and $\iota_1(r,k) = \iota_1(r,k) \forall k \in \mathcal{K}$ and $\forall n \in N_1 \cap N_2$, it holds that $\lambda_1(n) = \lambda_2(n)$ and $\iota_1(n,k) = \iota_1(n,k) \forall k \in \mathcal{K}$.

Definition 31 (Snapshot Graph). *Given a time interval $\tau = [\omega_o, \omega_c)$, a snapshot graph $\widetilde{G}_\tau$ (also $\widetilde{G}^{\omega_c}_{\omega_o}$) is the result of the union of all property graphs $G \in \widetilde{S}_\tau$ to a single property graph using the union operation of Definition 30. It holds:*

$$\widetilde{G}_\tau = \widetilde{G}^{\omega_c}_{\omega_o} = \bigcup_{G_i \in \widetilde{S}^{\omega_c}_{\omega_o}} G_i$$

Figure 7.2 shows the snapshot graph $\widetilde{G}^{15:45}_{14:45}$ resulting from coalescing the substream $\widetilde{S}^{15:45}_{14:45}$ highlighted with a red border in Figure 7.1.

In Section 7.3.2, we recall that clauses in a Cypher-query are functions that take a tableand output a table, potentially expanding the number of fields and adding new tuples. Similarly, in Seraph, we consider the time-based extensions of the notion above. In particular, a *time-varying table*, which is inspired by the time-varying relations from [26], generalizes the notion of the table into a function that maps the time Ω to a finite table. Moreover, we introduce *time-annotated tables* to extend Cypher's table with temporal boundaries.

Definition 32 (Time-annotated Table). *Given a time interval $\tau = [\omega_o, \omega_c)$, we define a time-annotated table $\widetilde{T}_\tau$ (also $\widetilde{T}^{\omega_o}_{\omega_c}$) as a bag or multiset of records $\tilde{\mu}$, where each is a partial function from names to values extended with names for the temporal annotations of the interval bounds ω_o and ω_c. Extending the convention used for Cypher's tables, we denote them as a tuple:*

$$\tilde{\mu} = (a_1 : v_1, \ldots, a_n : v_n, win\_start : \omega_o, win\_end : \omega_c)$$

where $a_1, \ldots, a_n$ are distinct names, and $v_1, \ldots, v_n$ are values. The names `win_start` *and* `win_end` *are reserved Keywords in Seraph justified by their use as identifiers for the window bounds, as per Definition 35. The order in which the fields appear is only for notation purposes. We refer to $dom(\tilde{\mu}) = \mathcal{A}$ as in Definition 26.*

For instance, Table 7.4 extends Table 7.2 with the aforementioned temporal annotations `win_start` and `win_end` with the values ω_o and ω_c, respectively.

Definition 33 (Time-varying Table). *Let $\widetilde{\mathcal{T}}$ be the set of all possible $\widetilde{T}$ in Ω. A time-varying table Ψ is a function that maps every time instant $\omega \in \Omega$ to a time-annotated Table $\widetilde{T} \in \widetilde{\mathcal{T}}$:*

$$\Psi : \Omega \to \widetilde{\mathcal{T}}$$

Given a time-varying table Ψ, we use the term $\Psi(\omega)$ to refer to the time-annotated table identified by the time-varying table at the given time instant ω. Moreover, we pose the following constraints on the definition of Ψ:

r.user_id	s.id	r.val_time	hops	win_start	win_end
1234	1	14:40	[2,3]	14:40	15:40
5678	2	14:58	[3,4]	14:40	15:40

Table 7.4: Time-annotated table as extension of Table 7.2.

- *Consistency, i.e., Ψ always identifies a time-annotated table.*

$$\Psi(\omega_i) = \widetilde{T}_{\omega_c}^{\omega_o} \text{ s.t. } \forall \mu \in \widetilde{T} : \mu.\omega_o \leq \omega_i < \mu.\omega_c$$

- *Chronologicality, i.e., Ψ always identifies the time-annotated table with the earliest (minimal) opening timestamp.*

$$\nexists\, \widetilde{T}_j \text{ s.t. } \mu_j \in \widetilde{T}_j, \mu_j.\omega_o \leq \omega_i < \mu_j.\omega_c$$

$$\forall \mu_i \in \Psi(\omega_i), \mu_j.\omega_o \leq \mu_i.\omega_o$$

- *Monotonicity, i.e., Ψ always identifies subsequent time-annotated tables for subsequent time instants.*

$$\forall \omega_i, \omega_j \text{ s.t. } \omega_i < \omega_j, \forall \mu_i \in \Psi(\omega_i)$$

$$\forall \mu_j \in \Psi(\omega_j), \mu_i.\omega_o < \mu_j.\omega_o \leq \mu_i.\omega_c < \mu_j.\omega_c$$

For instance, the time-annotated table shown in Table 7.4 would be identified by a given $\Psi(\omega_i)$ for any ω_i such that 14:40 $\leq \omega_i <$ 15:40. Based on the presented Seraph's data model, we are ready to define the query model in the next section.

7.5.2 Query Model

This section presents the query model of Seraph that extends Cypher to enable continuous queries over a property graph stream and thus satisfies the requirements **R2** and **R3**. Indeed, Cypher supports only one-time queries, which are evaluated once by the Cypher engine and whose result is a finite table. Seraph queries, on the other hand, are intended to be continuously evaluated until explicitly halted on a potential infinite input stream.

This paradigm-shift in the query execution model is named *Continuous Semantics*, i.e., processing an infinite input produces an infinite output [201]. Continuous semantics poses the challenge of formalizing a non-terminating evaluation. In practice, it implies that the result of a *continuous query* is the set of results that would be returned if the query would be executed at every time instant. Intuitively, if the objective computation is assumed to be stateless, continuous semantics can be achieved simply operating on each individual element in the input stream. In Seraph, this is the case for what concerns *data ingestion*. In fact, Cypher supports graph-based data ingestion by mapping elements of an input source, e.g., CSV, into property graphs. Similarly, Seraph ingestion operates on one event at time as shown in Listing 7.4, which is based on the Neo4j Kafka Connector [141].

On the other hand, the most common way to accomplish continuous semantics for stateful computations is via *snapshot reducibility*. In particular, we adapt the definition from [155], which in turn was adapted from [118], as follow:

```
LOAD STREAM FROM 'kafka:///bikes.stream' AS event
    MERGE (b:Bike {id: event.bike_id})
    MERGE (s:Station {id: event.location_id})
    CREATE (b)-[:rentedAt {val_time: event.time, user_id: event.uid}]->(s)
```

Listing 7.4: Example of graph-based ingestion in Seraph.

Definition 34 (Snapshot Reducibility). *Let S be a Property Graph Stream, CQ a continuous query, and Q its non-streaming counterpart. Snapshot reducibility states that each snapshot of the result of evaluating CQ over S is equivalent to applying Q over a snapshot of S, i.e.,*

$$\forall \omega_i \in \Omega, w = [\omega_o, \omega_c) \; s.t. \; \omega_o \leq \omega_i < \omega_c, CQ(S)_w == Q(S_w)$$

Snapshot reducibility induces the definition of operators, named *Windows*, that chunk the stream into finite snapshots for defining the evaluation scope. Several alternative window semantics exist [211]. In Seraph, we focus on time-based windows which operate according to the temporal annotation of the stream elements to define intervals that help select finite portions of the input stream. A time-based window is *deterministic*, iff the set of intervals it subsumes is independent of the timestamps of stream elements.

Definition 35 (Time-based window). *A time-based window $w = [\omega_o, \omega_c)$ is a time interval between a start time instant ω_o and an end time instant ω_c (exclusive). Let the triple (ω_0, α, β) be a window configuration, where*

- *ω_0 is the earliest timestamp defining the start of the first window instance,*
- *α is the window size (in time units), and*
- *β is the slide size (in time units), such that two consecutive windows overlap of at most α - β.*

A window operator $\mathcal{W}(\omega_0, \alpha, \beta)$ identifies a infinite set of windows:

$$\mathcal{W}(\omega_0, \alpha, \beta) = \left\{ w_i = [\omega_{o_i}, \omega_{c_i}) \;\middle|\; \begin{array}{l} i \in \mathbb{N}_0 \wedge \omega_{o_i} = \omega_0 + i\beta \wedge \\ \omega_{c_i} = \omega_0 + i\beta + \alpha \end{array} \right\}$$

If further holds that $|\omega_{o_i} - \omega_{c_i}| = \alpha$ and $\exists w_{i+1} s.t. |\omega_{o_i} - \omega_{o_{i+1}}| = \beta$, i.e., the distance of the lower bounds of two succeeding windows w_i and w_{i+1} is the sliding size β.

Applying a time-based window operator $\mathcal{W}$ to a Property Graph Stream S deterministically identifies an infinite set of substreams $\widetilde{\mathcal{S}}$, which lays the ground of continuous query execution.

$$\tilde{\mathcal{S}} = \mathcal{W}(\omega_0, \alpha, \beta)(S) = \left\{ \widetilde{S}_w | \forall w \in \mathcal{W}(\omega_0, \alpha, \beta) \right\}$$

As per their characterisation [25], continuous queries yield their results as if the queries were evaluated for every time instant. Since such an approach is impractical, stream processing engines typically control the execution by customising the reporting of results [57]. However, delegating the definition of the reporting to the internals of the engines has caused idiosyncrasies in the operational semantics [2] in the past that may lead equivalent queries to produce different results on different engines [53]. Moreover, declarative control of the query results reporting is a well-known stream processing desideratum [198]. To this extent, we define the sequence of *evaluation time instants* as follows.

Definition 36 (Evaluation time instants). *We define the sequence of time instants at which an evaluation of the query occurs as evaluation time instants. Such a sequence, namely ET, is potentially infinite. Notably, the sequence depends on the initial time instant*

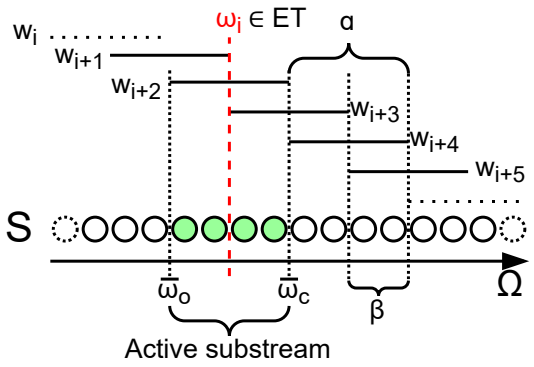


Figure 7.4: Selecting the active substream.

ω_0 and the slide size β, as defined in Definition 35. In particular, we define the ET sequence as follows:

$$ET = \{\omega \mid (\omega - \omega_0)/\beta = 0\}$$

For every $\omega_i \in ET$, a query evaluation is triggered. It is now necessary to identify the property graph substream $\widetilde{S}_\tau$ with $\omega_i \in \tau$, from which a snapshot graph $\widetilde{G}_\tau$ is constructed that is the input of the query evaluation. We refer to $\widetilde{S}_\tau$ as the *active substream*.

Definition 37 (Active Substream). *Given a time instant ω_i and the infinite set of all substreams $\tilde{\mathcal{S}}$, the active substream $\widetilde{S}_{\overline{w}}$ is the earliest property graph substream of all substreams that are valid at ω_i. I.e., it exists one window $\overline{w} = [\overline{\omega_o}, \overline{\omega_c}) \in \mathcal{W}(\omega_0, \alpha, \beta)$ such that $\omega_i \in \overline{w}$ and $\forall w = [\omega_o, \omega_c) \in \mathcal{W}(\omega_0, \alpha, \beta) : \overline{\omega_o} = min(\omega_o)$.*

For hopping time-based window operators (also denoted as *tumbling*), the identification is intuitive, since there is just one substream per time instant: $\widetilde{S}_{\overline{w}} = \widetilde{S}_\tau$ with $\overline{w} = \tau = [\omega_o, \omega_c) = [\overline{\omega_o}, \overline{\omega_c})$. For overlapping time-based window operators (also denoted as *sliding*), i.e., $\mathcal{W}(\omega_0, \alpha, \beta)$ s.t. $\beta < \alpha$, multiple substreams could be identified at each ω_i. In such scenario, we consider the one with earliest opening timestamp as defined above.

Figure 7.4 illustrates the identification of the active substream. One can see the set of windows $w_i, w_{i+1}, \ldots$ where each has the size α and a distance of β. The infinite property graph stream S is represented as multiple circles ◯. For a given evaluation time instant $\omega_i \in ET$, marked with a red dashed line, two windows exist that include this time instant: w_{i+2} and w_{i+3}. Note that $\omega_i \notin w_{i+1}$, since a window is a close-open interval excluding the upper interval bound. From both windows w_{i+2} and w_{i+3} we select the one with the earliest (smallest) lower interval bound ω_o as $\overline{w} = [\overline{\omega_o}, \overline{\omega_c})$. The substream $\widetilde{S}_{\overline{w}}$ is thus the active substream, whose property graphs are marked with green circles in the figure.

The union of all property graphs of $\widetilde{S}_{\overline{w}}$ results in a snapshot graph $\widetilde{G}_{\overline{w}}$, which we call *active snapshot graph*. On each evaluation time instant, the inner Cypher query is evaluated on the respective active snapshot graph. Each query evaluation results in a time-varying table Ψ that holds the tuples μ representing the time-annotated mappings of found matches. Finally, the continuous semantics implies an infinite output stream as result of a query evaluation. *Streaming operators*, as defined by Arasu et al. [19], reintroduce the temporal dimension in the data to construct a stream. We adapt this approach for creating a output stream of timestamped tuples (μ, ω) from the time-varying table $\Psi(\omega)$.

Definition 38 (Streaming Operators). *A streaming operator is defined by the pair (Ψ,ω), i.e., a time-varying table and a time instant, typically the current evaluation time instant $\omega_{now} \in ET$. We differentiate three streaming operators:*

- *The RStream outputs each tuple derived from Ψ timestamped with the evaluation time.*

$$RStream(\Psi, \omega) = \{(\mu, \omega) | \mu \in \Psi(\omega)\}$$

- *The IStream outputs the results that are part of the current evaluation result but are not in the previous one.*

$$IStream(\Psi, \omega_j, \omega_{j-1}) = \{(\mu, \omega_j) \mid \mu \in \Psi(\omega_j) \setminus \Psi(\omega_{j-1})\}$$

- *The DStream outputs the results that are part of the previous evaluation result but are not in the current one.*

$$DStream(\Psi, \omega_j, \omega_{j-1}) = \{(\mu, \omega_j) \mid \mu \in \Psi(\omega_{j-1}) \setminus \Psi(\omega_j)\}$$

After a detailed formalization of the query model in this section, the syntax and semantics of Seraph will be discussed in the following by bringing all the introduced concepts together.

7.5.3 Formal Syntax and Semantics

Seraph's key components are declarative (**R1**) *clauses* and *queries* that operate timely on the presented data model (Section 7.5.1) and query model (Section 7.5.2). From now on we will color Seraph syntax in green, leaving Cypher syntax in blue. We preserve the expressiveness of Cypher by defining only extensions, which satisfies requirement **R4**. In Figure 7.5, we illustrate the interactions and transitions from one component to another. In the upper left corner, we can see the input property graph stream, formally defined as

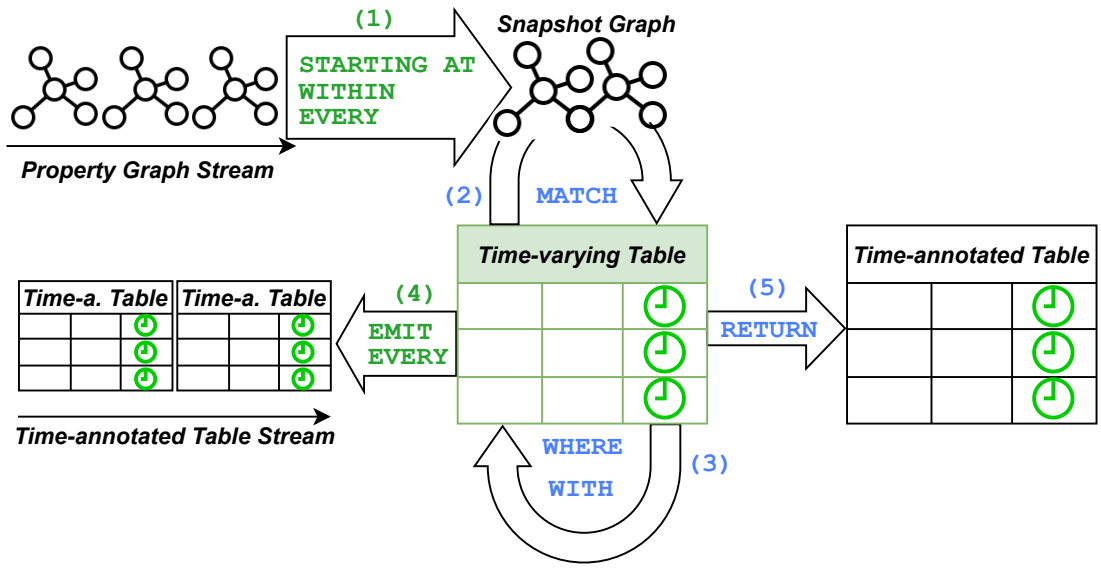


Figure 7.5: Seraph's data and query model Interaction.

S in Definition 28. The combination of three clauses (marked (1)), namely **STARTING AT**, **WITHIN** and **EVERY**, form the configuration of the window operator $\mathcal{W}$ from Definition 35. The window operator generates substreams $\widetilde{S}_\tau$ from the property graph stream S, each of which is combined into a snapshot graph $\widetilde{G}_\tau$.

The semantics of a **MATCH** clause is the pattern matching which takes as input a (initially empty) time-varying table Ψ, evaluates a pattern π matching on the snapshot graph $\widetilde{G}_\tau$, and in turn generates a time-varying table Ψ with extended set of field names and rows as output. In the figure, this is shown as a semicircular arrow marked with (2). The set of assignments that are the result of a **MATCH** clause as a time-varying table can be filtered via a selection using the **WHERE** clause (marked (3)) and thus again has a time-varying table as the result. Likewise, a projection of a time-varying table can be made via the **WITH** clause (also marked with (3)), which serves as input for another **MATCH** clause. This concept is taken from Cypher and allows the combination of several **MATCH** clauses.

In Seraph, the output of the evaluation result of one (or more) **MATCH** clauses can be emitted in two ways: a) as a stream of time-annotated tables $\widetilde{T}$ via the **EMIT** clause (marked with (4)) or b) as a single time-annotated table $\widetilde{T}$ via **RETURN** clause (marked with (5)). The former a) converts each time-varying table into a time-annotated table at each evaluation time ET using the projections specified by **EMIT** and the evaluation time instants specified by **EVERY**. It thus creates a stream of time-annotated tables. Second b) emits only one result. At the first evaluation time instant after the start time (defined by **STARTING AT**), the query is evaluated and the resulting time-varying table is converted into a time-annotated table using the projections specified by **RETURN**.

After this high-level overview, we can now present the formal syntax of a Seraph query, which is given in Figure 7.6. The semantics of expressions of Cypher, like values, variables, maps, lists etc., remain unchanged and can be derived from [71]. Furthermore, we provide a Seraph query parser open-source on GitHub [44].

Queries. The **REGISTER QUERY** clause allows for registering a new query with name $a \in \mathcal{A}$ into the system that implements Seraph. The name is used to identify the registered query and allows editing and deleting a previously registered query. The **STARTING AT** clause defines the first evaluation time instant, which is important for all window seman-

```
query^Srph ::= REGISTER QUERY a STARTING AT time { a ∈ A
               query^Δ
               stream_op
               EVERY range }
query^Δ    ::= RETURN ret | EMIT ret | clause^Δ query^Δ
clause^Δ   ::= MATCH pattern_tuple WITHIN range
               [WHERE expr]
               | WITH ret [WHERE expr] | UNWIND expr AS a
stream_op  ::= ON ENTERING | ON EXIT | SNAPSHOT
range      ::= <ISO_8601_duration>
time       ::= <ISO_8601_datetime>
ret        ::= * | expr [AS a] | | ret , expr [AS a]
```

Figure 7.6: Seraph's syntax based on Cypher's one in Figure 7.3.

tics, since from this points all windows are defined. This time instant, given as ISO8601 datetime, is used as the configuration ω_0 of the window operator from Definition 35, and is constant for the registered query.

The following body of the Seraph query is encapsulated by curly braces `{...}` and consists of three parts: the *query*, the *stream operator* and the *evaluation interval*. A *query* is a sequence of clauses ending with the `RETURN` or `EMIT` statement. Both contain the return list, which is either $*$, or a sequence of expressions, optionally followed by `AS a`, to provide their names. They define what to include in the query result set. The *stream operator* determines which streaming operator is used, which are defined in Definition 38. In particular, the `SNAPSHOT` clause specifies that the $RStream$ operator has to be used, while the `ON ENTERING` and `ON EXIT` clauses allow for selecting $IStream$ and $DStream$, respectively. The *evaluation interval*, i.e., sequence of evaluation time instances, can be specified using the `EVERY` clause, together with the `STARTING AT` clause. In particular, the `EVERY` clause defines the frequency of the evaluation, which can be specified with an ISO 8601 duration. The `STARTING AT` clause, instead, defines the first evaluation time instant as an ISO 8601 datetime.

Clauses. Seraph clauses are functions that take time-varying tables and produce time-varying tables. Analogous to Cypher, matching clauses are pattern matching statements of the form `MATCH pattern WITHIN range WHERE expr`, where `WHERE` is optional. The width parameter of windows is defined using the `WITHIN` clause, which is attached to every `MATCH` and its pattern definition. Thus, every pattern can be matched in its own window width. The `MATCH` clause extends the set of field names of $\Psi(\omega)$ by adding field names that correspond to names occurring in the pattern but not in μ. It also adds tuples to $\Psi(\omega)$, based on matches of the pattern that are found in the snapshot graph $\widetilde{G}_\tau$. Analogous to the Cypher definitions is `UNWIND` another clause that expands the set fields, and `WITH` clauses that can change the set of fields. In addition, `WITH` allows query parts to be chained together, piping the results from one to be used as starting points or criteria in the next.

Finally, we model the continuous evaluation process by including the evaluation time in the Cypher evaluation semantics. The continuous query answering is done by executing the query at each time instant of the sequence ET Given a fixed time instant, the operators can work in a time-agnostic way composing the semantics of Cypher in the one of Seraph. The semantics of queries associates a query SQ and a snapshot Property Graph $\widetilde{G}$ with a function $[\![SQ]\!]_{\widetilde{G}}$ that takes a time-varying table and a time instant and returns a time-varying table. The evaluation of a query starts with the time-varying table containing one empty tuple, which is then progressively changed by applying functions that provide the semantics of SQ's clauses. The composition of such functions, i.e., the semantics of SQ, is also a function, which defines the output as:

$$output(SQ, \widetilde{G}, \omega) = [[SQ]]_{\widetilde{G}}(\Psi, \omega)$$

This new concept requires a revision of the definitions of the existing Cypher evaluation of queries, clauses and expressions. We show all continuous evaluation semantics of redefined queries and clauses in Figure 7.7.

In particular, the semantics of the `MATCH` clause is described through the set $\overline{match}(\pi, \widetilde{G}, \mu)$, which is a redesign of the $match(\bar{\pi}, G, u)$ of Cypher (Section 7.3.2) with respect to the continuous evaluation semantic and the concept of snapshot graphs. Let π a path pattern,

$$
\begin{aligned}
[\![\texttt{RETURN}*]\!]_{\widetilde{G}}(\Psi,\omega) &= \Psi(\omega) \text{ where } \omega \in [\omega_o, \omega_c), \\
&\quad \text{and } \omega_o, \omega_c \text{ are the time annotations of } \Psi(\omega) \\
[\![\texttt{EMIT}*]\!]_{\widetilde{G}}(\Psi,\omega) &= \forall \omega_e \in ET\ [\![\texttt{RETURN}*]\!]_{\widetilde{G}}(\Psi,\omega_e) \text{ a proposal} \\
[\![\texttt{EMIT} * \texttt{ ON ENTERING}]\!]_{\widetilde{G}}(\Psi,\omega) &= [\![\texttt{EMIT}*]\!]_{\widetilde{G}}(\Psi,\omega),\ \Psi = \{\mu \mid \mu \in \Psi(\omega) \setminus \Psi(\omega-1)\} \\
[\![\texttt{EMIT} * \texttt{ ON EXIT}]\!]_{\widetilde{G}}(\Psi,\omega) &= [\![\texttt{EMIT}*]\!]_{\widetilde{G}}(\Psi,\omega),\ \Psi = \{\mu \mid \mu \in \Psi(\omega-1) \setminus \Psi(\omega)\} \\
[\![\texttt{EMIT} * \texttt{ SNAPSHOT}]\!]_{\widetilde{G}}(\Psi,\omega) &= [\![\texttt{EMIT}*]\!]_{\widetilde{G}}(\Psi,\omega),\ \Psi = \{\mu \mid \mu \in \Psi(\omega)\} \\
[\![\texttt{WITH}*]\!]_{\widetilde{G}}(\Psi,\omega) &= \Psi(\omega) \text{ if } \Psi(\omega) \text{ has at least one field} \\
[\![\texttt{WITH ret WHERE expr}]\!]_{\widetilde{G}}(\Psi,\omega) &= \Psi(\omega) \text{ if } \Psi(\omega) \text{ has at least one field} \\
[\![\texttt{STARTING AT } \omega_0 \texttt{ MATCH } \pi & \\
\texttt{WITHIN } \alpha \texttt{ EVERY } \beta]\!]_S &= [\![\texttt{MATCH}\pi]\!]_S^{W(\omega_0,\alpha,\beta)}(\Psi,\omega) \\
&\widehat{=}\ [\![\texttt{MATCH}\pi]\!]_{W(\omega_0,\alpha,\beta)(S)}(\Psi,\omega) \\
&\widehat{=}\ [\![\texttt{MATCH}\pi]\!]_{\widetilde{S}^{\omega_o}_{\omega_c}(\omega)}(\Psi,\omega) \\
&\widehat{=}\ [\![\texttt{MATCH}\pi]\!]_{\widetilde{G}_{\overline{w}}}(\Psi,\omega) \\
&= \biguplus_{\mu\in\Psi(\omega)}\{\mu\cdot\mu' | \mu' \in \overline{match}(\pi,\widetilde{G},\mu)\}
\end{aligned}
$$

Figure 7.7: Formal semantics of Seraph query and clauses.

$\widetilde{G}$ a snapshot graph, μ a time-annotated assignment, p a path in $\widetilde{G}$, π' a path pattern in the set of all rigid paths rigid(π) and $(p, \widetilde{G}, \mu * \mu') \vDash \pi'$ as satisfaction of π' in a path p in $\widetilde{G}$, the set of matches is defined as follows:

$$
\overline{match}(\pi,\widetilde{G},\mu) = \biguplus_{\substack{p\in\widetilde{G} \\ \pi'\in\text{rigid}(\pi)}} \left\{ \mu' \;\middle|\; \begin{array}{c} \text{dom}(\mu')=\text{free}(\pi)-\text{dom}(\mu) \\ \wedge (p,\widetilde{G},\mu*\mu')\vDash\pi' \end{array} \right\}
$$

For space reasons, we did not include the semantics of non-essential language components like `UNWIND`, `OPTIONAL`, renaming, and expressions. However, under the snapshot reducibility assumption [155], the continuous extension of such an operation is trivial. For a complete overview, we invite the interested reader to read the Cypher technical report [71].

7.5.4 Running Example vs Seraph

In this section we define a continuous query for the micromobility example of Section 7.2. With Seraph, the analytics team of RideAnywhere can register a continuous query that checks the rentals for student users applying the described trick of subsequent rentals. Lets discuss the Seraph query depicted in Listing 7.5, which continuously monitors the rentals for the pattern.

The `REGISTER QUERY` clause (line 1) allows for naming and registering the query into the system that manages Seraph queries. To define the time when the first evaluation will start, the `STARTING AT` clause is used, with a time instant. The query itself (line 2 to 12) is the query body that defines the pattern, its conditions, the projections, and the result emitting. Let us go through the query and compare it with the Cypher solution

```
1  REGISTER QUERY student_trick STARTING AT 2022-10-14T14:45 {
2    MATCH (:Bike)-[r:rentedAt]->(s:Station),
3          q = (b)-[:returnedAt|rentedAt*3..]-(o:Station)
4    WITHIN PT1H
5    WITH r, s, q, relationships(q) AS rels,
6      [n IN nodes(q) WHERE 'Station' IN labels(n) | n.id] AS hops
7    WHERE ALL(e IN rels WHERE
8      e.user_id = r.user_id AND e.val_time > r.val_time AND
9      (e.duration IS NULL OR e.duration < 20) )
10   EMIT r.user_id, s.id, r.val_time, hops
11   ON ENTERING
12   EVERY PT5M
13 }
```

Listing 7.5: Continuously retrieve users that use the free period for two subsequent rentals in the last hour using Seraph.

given in Listing 7.1. Since the desired window behavior is now natively supported by Seraph, we directly start by defining the desired pattern.

The `MATCH` clause defines the pattern π we are looking for (line 2 to 3). Note that compared to the previous Cypher query, the predicate applying the edge filtering for the window is obsolete. By `WITHIN` we define the width of the window for this pattern, which is 1 hour (`PT1H`). The predicates of the `WHERE` clause are equal to the ones of the Cypher query. Instead of the `RETURN` clause we use the `EMIT` clause to get a continuous stream of time-annotated tables and define the projected attributes for the resulting tuples enhanced with bounds of the current window that is built by Seraph (`win_start` and `win_end`). At line 11 the `ON ENTERING` operator allows for emitting only new matches entering the window, which satisfies requirement **R3**. The `EVERY` operator specifies the frequency of the evaluation process. Here, we define it as 5 minutes specified by `PT5M`. The operators `STARTING AT`, `WITHIN` and `EVERY` build the continuous evaluation and thus satisfy requirement **R2**. As Seraph only expands Cypher, we preserve the expressiveness and hence meet requirement **R4**.

To summarize, every 5 minutes starting from 14:45h, the system evaluates a pattern on the active snapshot graph defined by a 1h window and emits a stream of time-annotated tables, including the users that extend their rental time by using subsequent free rentals.

Let us analyze the output of the query at different time instants.

14:45h The 1h window covers only the outer left graph depicted in Figure 7.1. Just a bike was rented, which needs no notification.

14:46h - 14:59h The query emits no event.

15:00h The two left graphs in Figure 7.1 are merged. User `1234` returned a bike and rented one again. User `5678` rented a bike, too. However, the resulting snapshot graph is queried without any match.

15:01h - 15:14h The query emits no event.

15:15h The three left graphs in Figure 7.1 are in the active substream and thus merged to a snapshot graph, which leads to a match: user `1234` applying the trick. Since 15:15h is an evaluation time, the time-annotated table (Table 7.5) is emitted.

r.user_id	s.id	r.val_time	hops	win_start	win_end
1234	1	14:40	[2,3]	14:15	15:15

Table 7.5: Outputs of Seraph continuous query at 15:15h.

r.user_id	s.id	r.val_time	hops	win_start	win_end
5678	2	14:58	[3,4]	14:40	15:40

Table 7.6: Outputs of Seraph continuous query at 15:40h.

15:20h The fourth graph arrived with the information that user 5678 returned and rented again a bike.

15:21h - 15:39h The query emits no event.

15:40h All graphs of Figure 7.1 are in the active substream and thus unified to a snapshot graph (cf. Figure 7.2). Another match is found: user 5678 is applying the trick, too. The query's output at 15:40h is depicted in Table 7.6. Since we used `ON ENTERING`, just the new match, i.e., user 5678, is part of the resulting time-annotated table.

7.6 Implementation

Since this work provides the formal description of Seraph for paving the road to future implementations, we briefly discuss our plans in such a direction.

Graph Stream Processing (GSP) Engine. We built an proof of concept implementation [167] of a GSP engine with Seraph language support. It is open-source available under Apache-2.0 license and based on Neo4j and RSP4J [208], a library for fast-prototyping stream processing engines. A query parser [44] based on ANTLR is also available to validate the syntax design. Notably, this first POC has the goal of empirically proving Seraph's feasibility and enabling various tests and evaluations. In the short term, we also plan to test other Cypher-compatible embedded graph engines like Kuzu [104] or Memgraph [136].

We plan a first round of optimization focusing on query planning at different levels, including native operators and efficient window maintenance. We also plan to explore the adoption of advanced windowing as described in the recent survey [213], as well as optimizations regarding concurrent queries and avoidable re-executions on equal window contents.

Distributed GSP. In the medium term, we plan to explore a distributed implementation of Seraph based on a stream processing framework such as Apache Flink [37] or Apache Spark [225]. Here we can benefit from our work on Gradoop, which has integrated Cypher-based pattern matching based on Apache Flink.

A second round of optimization will focus on system-level investigation as in [86]. In particular, we will explore operator placement and fusions, graph stream partitioning, and distributed join algorithms.

7.7 Conclusion

In this section, we introduced Seraph, a declarative graph query language that compositionally enriches Cypher for handling streams of property graphs and for continuous query answering. In particular, it shows that Seraph is designed to overcome the limitations of Cypher for continuous processing: 1) Seraph's data model can represent streams of property graphs; 2) Seraph's query model allows continuous evaluation over Cypher semantics by creating snapshot graphs from the graph stream using windowing and evaluating the query under snapshot reducibility. In addition, 3) we demonstrated the capabilities of Seraph in three industrial use cases: network monitoring, real-time tracing, and bike sharing (our running example). The formal foundations we lay in this work will pave the way for future continuous graph query languages, such as a continuous extension to GQL, the ISO standard graph query language, whose first version is expected to appear in 2024.

Part IV

Epilogue

8 Lessons Learned from Gradoop

One of the main contributions of this dissertation is the Gradoop project, which aims to develop a comprehensive open-source framework for distributed processing and analysis of large temporal property graphs. We combined positive features of graph database systems and distributed graph processing systems and extended them in several ways, e.g., with support for logical graphs, graph collections, bitemporal maintenance, and built-in analysis capabilities, including structural graph transformations and temporal graph analysis operators. Given that this project has been running for over 8 years, we will now reflect on some of our design decisions regarding technology selection, data model, and operator concept, as well as a discussion on the usage and system acceptance.

8.1 Apache Flink

One of the first and most important design decisions was the selection of a suitable processing framework to enable the development of a comprehensive, extensible, and horizontally scalable graph analysis framework. At that time, Apache Flink was shortlisted and finally chosen because of its rich set of composable (Flink) transformations and support for automatic program optimization without needing more profound system knowledge.

In recent years, however, Apache Flink has focused on becoming a pure stream processing engine so that the DataSet API (used by Gradoop) is soft deprecated since version 1.12 [202]. We have, therefore, begun to evaluate alternate processing frameworks such as the DataStream and Table API of Apache Flink. Although a re-implementation of a large part of Gradoop and its operators would be necessary, the reorientation can also provide advantages for improving Gradoop's performance and feature set: The streaming model of Apache Flink already supports two different notions of time (processing- and event time, similar to the bitemporal model of the TPGM), watermarks for out-of-order streams, several window processing features and state backends to materialize intermediate results. Apache Flink has introduced a BATCH execution mode for the DataStream API, which optimizes the stream processing engine for finite data sets. Such a significant change could thus allow better support for processing, analysis, and continuous queries on graph streams [29] that we plan to address in the future.

8.2 Logical graphs and collections

A unique selling point of Gradoop's original EPGM and the temporal TPGM models is the introduction of logical graphs as an abstraction of a subgraph that can be easily semantically enhanced without changing the graph's structure by adding new vertex or edge types or adding redundant properties. Graph collections, i.e., sets of logical graphs, are a hugely valuable data structure for modeling (possibly overlapping) logical graphs. Graph collections also facilitate the use of binary graph operations [116], like intersection and union, an essential part of graph theory, for property graphs. In addition, they are used as a result of analytical operators that produce multiple graphs, for example, graph pattern matching, where each match represents a logical graph.

8.3 Operator concept

Another core design decision was the methodology to introduce *operators* that can be flexibly combined to define analytical workflows. Similar to the transformations between datasets in distributed processing engines, we developed single *analytical operators* that are closed over the model. The internal logic of an operator is a composition of (Flink) transformations hidden from the analyst, thus providing a top-level abstraction of the respective function.

The large number of operators offered, which contain both simple analyses and graph algorithms, can, therefore, be used as a toolkit for composing complex analysis pipelines. An analyst with programming experience can write new or modify existing operators (which requires knowledge of the implementation details) to extend the functional scope of Gradoop. This is possible through public Java interfaces for all types of operators.

Further, a typical feature of distributed in-memory systems is a scheduler that translates and optimizes the dataset transformations used in the program to a directed acyclic graph (DAG). Such transformations, represented by vertices in the DAG, are combined and chained to optimize the overall data flow. This, however, results in a disadvantage of the operator concept since the relation between transformations and operators often gets diluted. Consequently, performance issues with specific operators and their dataset transformations are complex to identify.

8.4 Temporal extensions

The evolution of entities and relationships is a natural characteristic of many graph datasets that represent a real-world scenario [187]. A static graph model like the PGM or the initial EPGM of Gradoop is, in most cases, unsuitable for performing analyses that specifically examine the development of the graph over time. We found that an extension of the data model and the respective operators (including new operators for solely temporal analysis) covered many requirements of frequently used temporal analysis, for example, the retrieval of a snapshot, without building a completely new model and prototypical implementation of a new framework.

One weakness identified was the insufficient support for frequent property value changes. To overcome this weakness of the data model, we have shown in Section 6.2 the

TPGM$^+$, a TPGM extension to lift the temporality also on the property level. In addition, through the operator concept, it is possible to combine static and temporal operators, for example, to first filter for entities and relationships of interest and then analyze their evolution with the grouping operator and its temporal aggregations.

In future work, we plan to develop alternatives to the GVE data layout (see Section 3.4) without separating vertices and edges so that TPGM integrity conditions can be checked more efficiently. We will also investigate separating the newest graph state from its history for increased performance.

8.5 Scalability

So far, evaluations have shown that Gradoop generally scales very well with increasing dataset sizes. On the other hand, when increasing the number of machines on a fixed dataset, the speedup reaches its limit relatively fast (for the considered workloads and datasets, the speedup mainly was lower than 10 with the default hash-based data partitioning). The main reason for this behavior can be seen in the strong dependency on the underlying Flink system and its optimizer and scheduler, which are not tailored to graph data processing.

Data distribution is by default based on a hash partitioning, particularly for intermediate results in an analytical pipeline, that prevents the utilization of data locality for our graph operators but can result in significant communication overhead even for traversing edges. Therefore, we prepared a possibility to partition a graph by label to reduce execution complexity for large-scale graphs. However, experience shows that a single partition strategy is not suited for all analytical operators Gradoop provides. Part of our future research will thus be to develop improved data distribution and load balancing techniques to achieve a better speedup behavior for single operators and analysis pipelines.

8.6 Usage and acceptance

Gradoop is an open-source (Apache License 2.0) research framework that has been co-developed by developers from industrial partners and many students within their bachelor, master, and Ph.D. theses, which led to a good number of publications. It has been used by us and others within different industrial collaborations [181] and applications and serves as the basis for other research projects, e.g. on knowledge graphs [185].

Furthermore, concepts of Gradoop operators have been adopted by companies. For example, the graph grouping operator from Neo4j's APOC library was inspired by Gradoop's grouping operator [143]. The implementation of Gradoop's pattern matching operator [109] as a proof of concept for the distributed execution of Cypher(-like) queries, directly influenced the development of Neo4j's Morpheus [140] project, which provides the OpenCypher grammar for Apache Spark by using its SQL DataFrame API.

To make it easier to start using Gradoop, we deploy the system weekly to the Maven Central Repository. Thus it can be used in own projects by solely adding a dependency without any additional installation effort. We provide further a "getting started" guideline and many example programs in the GitHub repository of Gradoop and its GitHub wiki [172] to support the usage.

The provided analytical language GrALa can be used in two ways, via Java API or KNIME extensions, to define an analytical program. Consequently, analysts with and without programming skills can use the system for graph analysis. Operators can be configured in many ways, and it can be challenging to find out which particular configuration should be used for the desired analysis result. Further, many possible combinations of the operators may also represent a challenge for the analyst. We, therefore, provide example operator configurations and detailed documentation in the GitHub wiki [172] to assist in finding the right combination and configuration.

The open-source Temporal Graph Explorer, introduced in Section 4.2, provides a web-based user interface to run three selected TPGM operators with all configuration possibilities. This allows a user to test how each configuration of an operator, such as temporal grouping, affects the result. Currently, the Temporal Graph Explorer is used for demonstration purposes in the Living Lab of ScaDS.AI Dresden/Leipzig. There, visitors of the lab can test temporal graph analyses via preconfigured operators. A temporal micromobility graph containing bicycle rentals from Leipzig is provided as a data set.

9 Conclusion and Outlook

This dissertation gives a comprehensive overview of the research on the scalable management and analysis of temporal property graphs and query languages for graph streams. We now summarize the contributions of this dissertation in Section 9.1 and give an outlook on follow-up work in Section 9.2.

9.1 Conclusion

Graph databases, graph processing, and graph data science help provide insight into complex networks by providing data models in which entities and their relationships can be effectively modeled and analyzed. Gartner predicts that graph technology will be used in 80 percent of data and analytics developments by 2025, up from 10 percent in 2021 [21]. However, graph modeling, querying, and analysis often neglect one significant dimension: **time**.

Almost every real-world graph evolves over time, and information about this evolution is typically available but not natively maintained by graph systems. However, this has changed somewhat in recent years, at least in research. There is an increasing focus on dynamic graphs, especially temporal graphs and graph streams.

This dissertation joins the current research on temporal graphs and graph streams and addresses most challenges of this research field discussed in Section 1.1. The contribution of this work can be summed up in the following.

One framework to analyze them all. True to this motto, the graph analysis system Gradoop, which previously only supported static property graphs, has been comprehensively extended to process temporal property graphs from arbitrary domains. Existing systems specializing in the analysis of temporal graphs usually pursue only a specific analysis goal, e.g., querying using an extended language, developing a temporal graph algorithm, or the computation of temporal vertex metrics, or are specialized for a concrete use case. Gradoop, in contrast, is a general-purpose tool, not focused on any specific analysis or use case. The unique selling point of Gradoop is its flexible operator concept, which allows one to define arbitrary simple or complex processing and analysis pipelines and execute them in a distributed fashion. The graph model, called TPGM,

which includes the data model, operators, algorithms, and a declarative query language called GrALa, supports bitemporal versioning of nodes and edges, as well as abstractions of subgraphs, called logical graphs, which are also bitemporally managed and therefore called temporal graphs. Nodes and edges can be part of any number of temporal graphs, and temporal graphs can be part of graph collections. An operator of the TPGM is closed over the model, i.e., it consumes one or two temporal graphs or graph collections and outputs a temporal graph, a graph collection, or a generic dataset. This allows the flexible construction of analysis pipelines by concatenating operators.

In this work, new and extended operators that are specifically tailored to temporal graphs are proposed. For example, the snapshot operator extracts a graph from the graph history at a given time. The temporal grouping operator summarizes a complex temporal graph in an arbitrarily simple way, revealing hidden patterns in the temporal evolution of the graph. The temporal pattern matching operator has been provided with a query language called TemporalGDL, which allows, e.g., to order patterns and paths within the query chronologically or to compare intervals with an established algebra. Using real and artificial temporal graphs, Gradoop was evaluated to see how it reacts to increasing dataset size, cluster size, or changes in operator configurations. Although there is potential for optimization in the data representation and in the implementation of the operators, a good horizontal scalability of the system was demonstrated. Critical design decisions, such as the choice of framework, the operator concept, or the temporal extension, were evaluated in a lessons learned chapter at the end of the dissertation.

The usability of Gradoop and its data model was further demonstrated in two applications. First, the ability to combine the operators was demonstrated using a call center use case, where subgraph, snapshot, and grouping with roll-up functionality were intelligently combined to answer a complex analytical question. Second, a demonstration application called Temporal Graph Explorer (TGE) was presented. The web-based application allows the user to try out 3 operators of the TPGM. The resulting graph is visualized in the TGE after selecting an operator, its configuration, and a temporal input graph. The output is displayed on an interactive map if geographic information is available at the nodes. This allows the user to explore the graph and its evolution over time.

The focus then turned to nodes and graph metrics. It was shown that the static view of a metric is insufficient for temporal graphs since the graph evolves, and thus, the metrics also change. We show that a metric representing a scalar value for static graphs is a time series for temporal graphs. For various metrics based on vertex degree, temporal and evolutionary versions are formally defined and demonstrated with examples. After presenting a baseline algorithm for computing the degree evolution of all nodes of a temporal graph, a reference implementation in Gradoop was presented, and its runtime and scalability were evaluated using different temporal graph datasets. We showed that the implementation scales well even for graphs with more than 460 million nodes.

Gradoop is thus an open-source system that can be used to process temporal graphs in a distributed manner, combining modular operators as needed to create analysis workflows focused on the evolution of the graph.

Continuous querying and streams of graphs. The second central part of this work focuses on the continuous evaluation of temporal graphs and graph streams. A cooperation project with Oracle Labs showed that temporal graphs are suitable for

organizing time series data in the IoT domain, i.e., for modeling relationships between entities that produce time series like sensors. Since the TPGM does not support frequent changes in property values, it was necessary to extend it to an extended version, namely TPGM$^+$.

A temporal graph query language, called TPGQL, was also developed for the resulting TPGM$^+$. Based on Oracle's PGQL language, several temporal extensions were developed to allow, for example, finding temporal patterns in the graph at different points in time. To perform this type of evaluation continuously, a draft notification mechanism called Continuous Graph Notification (CGN) was also presented, which informs a recipient that a change in the graph may lead to a change in the query result. This contribution was a first step towards 1) combining temporal graphs and time series, which leads to a new research project called HyGraph, and 2) continuous query capabilities on changing graphs.

Continuing this research direction, a declarative graph query language for querying streams of property graphs was developed. The language, called Seraph, is based on the well-known Cypher graph query language - a base of the upcoming GQL standard. It was shown how an existing language like Cypher can be extended syntactically and semantically to allow for window-based evaluation of paths and patterns.

The formal definition of Seraph includes a data model that combines the property graph model with data streams, a query model with complete control over reporting, and the syntax and semantics of time-based operators that operate on the data model. The use of Seraph was demonstrated using a bike-sharing example and two other real-world use cases. As the new property graph query language GQL is being standardized, the work we contributed with Seraph is a foundation for extensions to this standard in the context of graph streams and continuous query evaluation.

9.2 Outlook

In the future, we will investigate how the TPGM and its operator concept can be realized by alternate technologies, e.g., using a distributed streaming model or Actor-like abstractions such as Stateful Functions [7], to process temporal graphs. Another area that we will work more on is to extend Gradoop to realize temporal knowledge graphs integrating data from different sources and continuously evolving the integrated data [88]. A further extension is the addition of layout algorithms to visualize temporal graphs and graph collections to offload the expensive calculation from front-end applications. Through our work on temporal degree metrics, we have also laid the foundation for the temporal extension of other static metrics, such as a temporal variant of betweenness centrality or clustering coefficient. Finally, we will investigate the overall performance optimization of Gradoop and add temporal graph algorithms as further operators to the framework.

The continuous analysis of graph streams is an emerging research area [36] and will continue to be considered in our future research. We already working on adaptations of TPGM operators for property graph streams, e.g., the so-called *Graph Stream Zoomer* [42], a window-based graph stream grouping system based on Apache Flink, offers the same grouping functionalities as the time-dependent grouping operator of the TPGM, only using graph streams as data model. With the work on Seraph (Chapter 7), we have already defined the semantics of a corresponding continuous query language. Our formal

definitions can be used to sketch a continuous extension to GQL, the upcoming ISO standard graph query language. In addition to the implementation plans of Seraph, we will explore i) how to query multiple streams simultaneously, ii) how to partition a property graph stream into logical substreams, and iii) how to incorporate static graph data into the continuous computation. Finally, we plan to vi) focus on graph-to-graph transformations as in GQL. Besides, we will integrate machine learning (ML) methods on temporal and streaming graphs, e.g., to identify diminishing and recurring patterns or predict further evolution steps.

Our research also continues in a project called *HyGraph* [43] funded by the German Research Foundation (DFG) and the French National Research Agency (ANR). The project already started (2023) and aims to design a hybrid data model that combines temporal graphs, time series, and graph streams. Among other things, the project will apply the operator concept of the TPGM to the hybrid data model to achieve the construction of analytical workflows.

Bibliography

[1] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. "Streaming Graph Partitioning: An Experimental Study". In: *Proc. VLDB Endow.* 11.11 (2018), pp. 1590–1603. DOI: 10.14778/3236187.3236208.

[2] Lorenzo Affetti, Riccardo Tommasini, Alessandro Margara, Gianpaolo Cugola, and Emanuele Della Valle. "Defining the execution semantics of stream processing engines". In: *J. Big Data* 4 (2017), p. 12. DOI: 10.1186/s40537-017-0072-9.

[3] Charu C. Aggarwal. "Extracting Real-Time Insights from Graphs and Social Streams". In: *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval, SIGIR 2018, Ann Arbor, MI, USA, July 08-12, 2018*, ed. by Kevyn Collins-Thompson et al. ACM, 2018, p. 1339. DOI: 10.1145/3209978.3210212.

[4] Charu C. Aggarwal and Karthik Subbian. "Evolutionary Network Analysis: A Survey". In: *ACM Comput. Surv.* 47.1 (2014), 10:1–10:36. DOI: 10.1145/2601412.

[5] Amir Aghasadeghi, Vera Zaychik Moffitt, Sebastian Schelter, and Julia Stoyanovich. "Zooming Out on an Evolving Graph". In: *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, ed. by Angela Bonifati et al. OpenProceedings.org, 2020, pp. 25–36. DOI: 10.5441/002/EDBT.2020.04.

[6] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. "Efficient pattern matching over event streams". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, ed. by Jason Tsong-Li Wang. ACM, 2008, pp. 147–160. DOI: 10.1145/1376616.1376634.

[7] Adil Akhter, Marios Fragkoulis, and Asterios Katsifodimos. "Stateful Functions as a Service in Action". In: *Proc. VLDB Endow.* 12.12 (2019), pp. 1890–1893. DOI: 10.14778/3352063.3352092.

[8] Sinan G. Aksoy, Emilie Purvine, Eduardo Cotilla Sanchez, and Mahantesh Halappanavar. "A generative graph model for electrical infrastructure networks". In: *J. Complex Networks* 7.1 (2019), pp. 128–162. DOI: 10.1093/comnet/cny016.

[9] Alexander Alexandrov et al. "The Stratosphere platform for big data analytics". In: *VLDB J.* 23.6 (2014), pp. 939–964. DOI: 10.1007/S00778-014-0357-Y.

[10] James F. Allen. “Maintaining Knowledge about Temporal Intervals”. In: *Commun. ACM* 26.11 (1983), pp. 832–843. DOI: 10.1145/182.358434.

[11] Anastasia Analyti and Ioannis Pachoulakis. “A survey on models and query languages for temporally annotated RDF”. In: *International Journal of Advanced Computer Science & Applications* 1.3 (2008), pp. 28–35.

[12] Landy Andriamampianina, Franck Ravat, Jiefu Song, and Nathalie Vallès-Parlangeau. “Semantic Centrality for Temporal Graphs”. In: *New Trends in Database and Information Systems - ADBIS 2023 Short Papers, Doctoral Consortium and Workshops: AIDMA, DOING, K-Gals, MADEISD, PeRS, Barcelona, Spain, September 4-7, 2023, Proceedings*, ed. by Alberto Abelló et al. Vol. 1850. Communications in Computer and Information Science. Springer, 2023, pp. 163–173. DOI: 10.1007/978-3-031-42941-5_15.

[13] Renzo Angles. “A Comparison of Current Graph Database Models”. In: *Workshops Proceedings of the IEEE 28th International Conference on Data Engineering, ICDE 2012, Arlington, VA, USA, April 1-5, 2012*, ed. by Anastasios Kementsietsidis et al. IEEE Computer Society, 2012, pp. 171–177. DOI: 10.1109/ICDEW.2012.31.

[14] Renzo Angles. “The Property Graph Database Model”. In: *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018*, ed. by Dan Olteanu et al. Vol. 2100. CEUR Workshop Proceedings. CEUR-WS.org, 2018.

[15] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. “Foundations of Modern Query Languages for Graph Databases”. In: *ACM Comput. Surv.* 50.5 (2017), 68:1–68:40. DOI: 10.1145/3104031.

[16] Renzo Angles et al. “G-CORE: A Core for Future Graph Query Languages”. In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, ed. by Gautam Das et al. ACM, 2018, pp. 1421–1432. DOI: 10.1145/3183713.3190654.

[17] Renzo Angles et al. “PG-Keys: Keys for Property Graphs”. In: *SIGMOD ’21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, ed. by Guoliang Li et al. ACM, 2021, pp. 2423–2436. DOI: 10.1145/3448016.3457561.

[18] Renzo Angles et al. “PG-Schema: Schemas for Property Graphs”. In: *Proc. ACM Manag. Data* 1.2 (2023), 198:1–198:25. DOI: 10.1145/3589778.

[19] Arvind Arasu, Shivnath Babu, and Jennifer Widom. “The CQL continuous query language: semantic foundations and query execution”. In: *VLDB J.* 15.2 (2006), pp. 121–142. DOI: 10.1007/s00778-004-0147-z.

[20] Nosratali Ashrafi-Payaman, Mohammadreza Kangavari, Saeid Hosseini, and Amir Mohammad Fander. "GS4: Graph stream summarization based on both the structure and semantics". In: *J. Supercomput.* 77.3 (2021), pp. 2713–2733. DOI: 10.1007/s11227-020-03290-2.

[21] Eric Avidon. *Gartner predicts exponential growth of graph technology*. 2021. URL: https://www.techtarget.com/searchbusinessanalytics/news/252507769/Gartner-predicts-exponential-growth-of-graph-technology (visited on 10/10/2023).

[22] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. "Models and Issues in Data Stream Systems". In: *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, ed. by Lucian Popa et al. ACM, 2002, pp. 1–16. DOI: 10.1145/543613.543615.

[23] Shivnath Babu and Jennifer Widom. "Continuous Queries over Data Streams". In: *SIGMOD Rec.* 30.3 (2001), pp. 109–120. DOI: 10.1145/603867.603884.

[24] Irina Balaur, Alexander Mazein, Mansoor Saqi, Artem Lysenko, Christopher J Rawlings, and Charles Auffray. "Recon2Neo4j: applying graph database technologies for managing comprehensive genome-scale networks". In: *Bioinformatics* 33.7 (Dec. 2016), pp. 1096–1098. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btw731. eprint: https://academic.oup.com/bioinformatics/article-pdf/33/7/1096/49038470/bioinformatics_33_7_1096.pdf.

[25] Daniel Barbará. "The Characterization of Continuous Queries". In: *Int. J. Cooperative Inf. Syst.* 8.4 (1999), p. 295. DOI: 10.1142/S0218843099000150.

[26] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth L. Knowles. "One SQL to Rule Them All - an Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables". In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, ed. by Peter A. Boncz et al. ACM, 2019, pp. 1757–1772. DOI: 10.1145/3299869.3314040.

[27] Jon Louis Bentley. "Multidimensional Binary Search Trees Used for Associative Searching". In: *Commun. ACM* 18.9 (1975), pp. 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007.

[28] Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Kilian Thiel, and Bernd Wiswedel. "KNIME - the Konstanz information miner: version 2.0 and beyond". In: *SIGKDD Explor.* 11.1 (2009), pp. 26–31. DOI: 10.1145/1656274.1656280.

[29] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. "Practice of Streaming Processing of Dynamic Graphs: Concepts, Models, and Systems". In: *IEEE Trans. Parallel Distributed Syst.* 34.6 (2023), pp. 1860–1876. DOI: 10.1109/TPDS.2021.3131677.

[30] Angela Bonifati, Stefania-Gabriela Dumbrava, Emile Martinez, Fatemeh Ghasemi, Malo Jaffré, Pacome Luton, and Thomas Pickles. "DiscoPG: Property Graph Schema Discovery and Exploration". In: *Proc. VLDB Endow.* 15.12 (2022), pp. 3654–3657. DOI: 10.14778/3554821.3554867.

[31] Karsten M. Borgwardt, Hans-Peter Kriegel, and Peter Wackersreuther. "Pattern Mining in Frequent Dynamic Subgraphs". In: *Proceedings of the 6th IEEE International Conference on Data Mining (ICDM 2006), 18-22 December 2006, Hong Kong, China*. IEEE Computer Society, 2006, pp. 818–822. DOI: 10.1109/ICDM.2006.124.

[32] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura M. Haas, Renée J. Miller, and Nesime Tatbul. "SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems". In: *Proc. VLDB Endow.* 3.1 (2010), pp. 232–243. DOI: 10.14778/1920841.1920874.

[33] Antal Buza. "Extension of CQL over Dynamic Databases". In: *J. UCS* 12.9 (2006), pp. 1165–1176. DOI: 10.3217/jucs-012-09-1165.

[34] Alexander Campos, Jorge Mozzino, and Alejandro A. Vaisman. "Towards Temporal Graph Databases". In: CEUR Workshop Proceedings 1644 (2016), ed. by Reinhard Pichler et al.

[35] Andrea Capocci, Vito Domenico Pietro Servedio, Francesca Colaiori, Luciana S. Buriol, Debora Donato, Stefano Leonardi, and Guido Caldarelli. "Preferential attachment in the growth of social networks: the case of Wikipedia". In: *CoRR* abs/physics/0602026 (2006). arXiv: physics/0602026.

[36] Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, and Asterios Katsifodimos. "Beyond Analytics: The Evolution of Stream Processing Systems". In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, ed. by David Maier et al. ACM, 2020, pp. 2651–2658. DOI: 10.1145/3318464.3383131.

[37] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. "Apache Flink™: Stream and Batch Processing in a Single Engine". In: *IEEE Data Eng. Bull.* 38.4 (2015), pp. 28–38.

[38] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. "Time-varying graphs and dynamic networks". In: *Int. J. Parallel Emergent Distributed Syst.* 27.5 (2012), pp. 387–408. DOI: 10.1080/17445760.2012.668546.

[39] Arnaud Casteigts, Kitty Meeks, George B. Mertzios, and Rolf Niedermeier. "Temporal Graphs: Structure, Algorithms, Applications (Dagstuhl Seminar 21171)". In: *Dagstuhl Reports* 11.3 (2021), pp. 16–46. DOI: 10.4230/DagRep.11.3.16.

[40] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. "Bigtable: A Distributed Storage System for Structured Data". In: *ACM Trans. Comput. Syst.* 26.2 (2008), 4:1–4:26. DOI: 10.1145/1365815.1365816.

[41] 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: *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, ed. by Pascal Felber et al. ACM, 2012, pp. 85–98. DOI: 10.1145/2168836.2168846.

[42] Christopher Rost and Max Zimmer and Rana Noureddin. *GitHub Repository - Graph Stream Zoomer*. 2023. URL: https://github.com/dbs-leipzig/graph-stream-zoomer (visited on 08/10/2023).

[43] Christopher Rost and Mouna Ammar. *HyGraph - A project combining temporal graphs with time-series and graph streams*. 2023. URL: https://hygraph.net (visited on 08/10/2023).

[44] Christopher Rost and Riccardo Tommassini and Maximilian Zimmer and Julian Pielmaier. *Seraph Query Parser*. 2023. URL: https://github.com/dbs-leipzig/seraph-parser (visited on 08/10/2023).

[45] Martino Ciaperoni, Edoardo Galimberti, Francesco Bonchi, Ciro Cattuto, Francesco Gullo, and Alain Barrat. "Relevance of temporal cores for epidemic spread in temporal networks". In: *Scientific reports* 10.1 (2020), pp. 1–15. DOI: 10.1038/s41598-020-69464-3.

[46] Milrose Consultants. *NYC 2018 Summer Streets Construction Embargo: July 30th - August 18th*. 2022. URL: https://www.milrose.com/insights/2018-summer-streets-construction-embargo (visited on 10/10/2023).

[47] Oracle Corp. *Oracle's Graph Database*. URL: https://www.oracle.com/de/database/graph/ (visited on 08/03/2023).

[48] Leonardo Cotta, Christopher Morris, and Bruno Ribeiro. "Reconstruction for Powerful Graph Representations". In: *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, ed. by Marc'Aurelio Ranzato et al. 2021, pp. 1713–1726.

[49] Michael Curtiss et al. “Unicorn: A System for Searching the Social Graph”. In: *Proc. VLDB Endow.* 6.11 (2013), pp. 1150–1161. DOI: 10.14778/2536222.2536239.

[50] C. J. Date. “Some Principles of Good Language Design (with especial reference to the design of database languages)”. In: *SIGMOD Rec.* 14.3 (1984), pp. 1–7. DOI: 10.1145/984549.984550.

[51] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: vol. 51. 1. 2008, pp. 107–113. DOI: 10.1145/1327452.1327492.

[52] Ariel Debrouvier, Eliseo Parodi, Matías Perazzo, Valeria Soliani, and Alejandro A. Vaisman. “A model and query language for temporal graph databases”. In: *VLDB J.* 30.5 (2021), pp. 825–858. DOI: 10.1007/S00778-021-00675-4.

[53] Daniele Dell’Aglio, Jean-Paul Calbimonte, Marco Balduini, Óscar Corcho, and Emanuele Della Valle. “On Correctness in RDF Stream Processor Benchmarking”. In: *The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II*, ed. by Harith Alani et al. Vol. 8219. Lecture Notes in Computer Science. Springer, 2013, pp. 326–342. DOI: 10.1007/978-3-642-41338-4_21.

[54] Daniele Dell’Aglio, Emanuele Della Valle, Jean-Paul Calbimonte, and Óscar Corcho. “RSP-QL Semantics: A Unifying Query Model to Explain Heterogeneity of RDF Stream Processing Systems”. In: *Int. J. Semantic Web Inf. Syst.* 10.4 (2014), pp. 17–44. DOI: 10.4018/ijswis.2014100102.

[55] Alin Deutsch et al. “Graph Pattern Matching in GQL and SQL/PGQ”. In: *SIGMOD ’22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, ed. by Zachary G. Ives et al. ACM, 2022, pp. 2246–2258. DOI: 10.1145/3514221.3526057.

[56] Reinhard Diestel. *Graph Theory, 4th Edition*. Vol. 173. Graduate texts in mathematics. Springer, 2012. ISBN: 978-3-642-14278-9.

[57] Nihal Dindar, Nesime Tatbul, Renée J. Miller, Laura M. Haas, and Irina Botan. “Modeling the execution semantics of stream processing engines with SECRET”. In: *VLDB J.* 22.4 (2013), pp. 421–446. DOI: 10.1007/s00778-012-0297-3.

[58] David Ediger, Robert McColl, E. Jason Riedy, and David A. Bader. “STINGER: High performance data structure for streaming graphs”. In: *IEEE Conference on High Performance Extreme Computing, HPEC 2012, Waltham, MA, USA, September 10-12, 2012*. IEEE, 2012, pp. 1–5. DOI: 10.1109/HPEC.2012.6408680.

[59] Maurice Eisenblätter. *Pull Request: Apache Parquet Source and Sink for Gradoop*. 2023. URL: https://github.com/dbs-leipzig/gradoop/pull/1586 (visited on 12/10/2023).

[60] John Ellson, Emden R. Gansner, Yifan Hu, and Stephen North. *DOT Language*. 2023. URL: https://www.graphviz.org/doc/info/lang.html (visited on 08/10/2023).

[61] Benjamin Erb, Dominik Meißner, Jakob Pietron, and Frank Kargl. "Chronograph: A Distributed Processing Platform for Online and Batch Computations on Event-sourced Graphs". In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, June 19-23, 2017*. ACM, 2017, pp. 78–87. DOI: 10.1145/3093742.3093913.

[62] Wei Fan, Meng Liu, and Yong Liu. "A Dynamic Heterogeneous Graph Perception Network with Time-Based Mini-Batch for Information Diffusion Prediction". In: *Database Systems for Advanced Applications - 27th International Conference, DASFAA 2022, Virtual Event, April 11-14, 2022, Proceedings, Part I*, ed. by Arnab Bhattacharya et al. Vol. 13245. Lecture Notes in Computer Science. Springer, 2022, pp. 604–612. DOI: 10.1007/978-3-031-00123-9_49.

[63] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. "On graph problems in a semi-streaming model". In: *Theor. Comput. Sci.* 348.2-3 (2005), pp. 207–216. DOI: 10.1016/j.tcs.2005.09.013.

[64] Santo Fortunato and Andrea Lancichinetti. "Community detection algorithms: a comparative analysis: invited presentation, extended abstract". In: (2009), ed. by Giovanni Stea et al., p. 27. DOI: 10.4108/ICST.VALUETOOLS2009.8046.

[65] Apache Software Foundation. *Apache HBase*. 2018. URL: https://hbase.apache.org/ (visited on 08/15/2023).

[66] Apache Software Foundation. *Apache Parquet*. 2023. URL: https://parquet.apache.org (visited on 12/10/2023).

[67] The Apache Software Foundation. *Apache Accumulo*. 2020. URL: https://accumulo.apache.org/ (visited on 08/01/2023).

[68] The Apache Software Foundation. *Apache ECharts - An Open Source JavaScript Visualization Library*. 2023. URL: https://echarts.apache.org/en/index.html (visited on 10/10/2023).

[69] The Apache Software Foundation. *CovidGraph – A Covid-19 Knowledge Graph*. 2023. URL: https://healthecco.org/covidgraph/ (visited on 08/01/2023).

[70] The Linux Foundation. *JanusGraph: an open source, distributed graph database*. 2022. URL: https://janusgraph.org (visited on 08/10/2023).

[71] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andres Taylor. "Cypher: An Evolving Query Language for Property Graphs". In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, ed. by Gautam Das et al. ACM, 2018, pp. 1433–1445. DOI: 10.1145/3183713.3190657.

[72] Max Franz, Christian Lopes, Dylan Fong, Mike Kucera, and Gary Bader. *Cytoscape.js - Graph theory (network) library for visualisation and analysis*. 2023. URL: https://js.cytoscape.org (visited on 10/10/2023).

[73] Linton C Freeman. "Centrality in social networks conceptual clarification". In: *Social Networks* 1.3 (1978), pp. 215–239.

[74] Swapnil Gandhi and Yogesh Simmhan. "An Interval-centric Model for Distributed Computing over Temporal Graphs". In: *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 2020, pp. 1129–1140. DOI: 10.1109/ICDE48307.2020.00102.

[75] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. "A Technique for Drawing Directed Graphs". In: *IEEE Trans. Software Eng.* 19.3 (1993), pp. 214–230. DOI: 10.1109/32.221135.

[76] Dieter Gawlick. *Temporal Property Graphs as Organizing Principles*. 2022. URL: https://labs.oracle.com/pls/apex/f?p=94065:15:115937665129700:2502 (visited on 10/10/2023).

[77] *Gelly: Flink Graph API*. 2023. URL: https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/gelly/ (visited on 10/10/2022).

[78] Kevin Gómez, Matthias Täschner, M. Ali Rostami, **Christopher Rost**, and Erhard Rahm. "Graph Sampling with Distributed In-Memory Dataflow Systems". In: *Datenbanksysteme für Business, Technologie und Web (BTW 2021), 19. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 13.-17. September 2021, Dresden, Germany, Proceedings*, ed. by Kai-Uwe Sattler et al. Vol. P-311. LNI. Gesellschaft für Informatik, Bonn, 2021, pp. 303–312. ISBN: 978-3-88579-705-0. DOI: 10.18420/BTW2021-15.

[79] Maoguo Gong, Ling-Jun Zhang, Jingjing Ma, and Licheng Jiao. "Community Detection in Dynamic Social Networks Based on Multiobjective Immune Algorithm". In: *J. Comput. Sci. Technol.* 27.3 (2012), pp. 455–467. DOI: 10.1007/S11390-012-1235-Y.

[80] Xiangyang Gou, Lei Zou, Chenxingyu Zhao, and Tong Yang. "Fast and Accurate Graph Stream Summarization". In: *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 2019, pp. 1118–1129. DOI: 10.1109/ICDE.2019.00103.

[81] Fabio Grandi. "T-SPARQL: A TSQL2-like Temporal Query Language for RDF". In: *Local Proceedings of the Fourteenth East-European Conference on Advances in Databases and Information Systems, Novi Sad, Serbia, September 20-24, 2010*, ed. by Mirjana Ivanovic et al. Vol. 639. CEUR Workshop Proceedings. CEUR-WS.org, 2010, pp. 21–30.

[82] Jonathan L. Gross and Jay Yellen, eds. *Handbook of Graph Theory*. Discrete Mathematics and Its Applications. Chapman & Hall / Taylor & Francis, 2003. ISBN: 978-1-58488-090-5. DOI: 10.1201/9780203490204.

[83] Hassan Halawa and Matei Ripeanu. "Position paper: bitemporal dynamic graph analytics". In: *GRADES-NDA '21: Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Virtual Event, China, 20 June 2021*, ed. by Vasiliki Kalavri et al. ACM, 2021, 7:1–7:12. DOI: 10.1145/3461837.3464514.

[84] Thomas Hartmann, François Fouquet, Matthieu Jimenez, Romain Rouvoy, and Yves Le Traon. "Analyzing Complex Data in Motion at Scale with Temporal Graphs". In: *The 29th International Conference on Software Engineering and Knowledge Engineering, Wyndham Pittsburgh University Center, Pittsburgh, PA, USA, July 5-7, 2017*, ed. by Xudong He. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2017, pp. 596–601. DOI: 10.18293/SEKE2017-048.

[85] Martin Hirzel, Guillaume Baudart, Angela Bonifati, Emanuele Della Valle, Sherif Sakr, and Akrivi Vlachou. "Stream Processing Languages in the Big Data Era". In: *SIGMOD Rec.* 47.2 (2018), pp. 29–40. DOI: 10.1145/3299887.3299892.

[86] Martin Hirzel, Robert Soulé, Scott Schneider, Bugra Gedik, and Robert Grimm. "A catalog of stream processing optimizations". In: *ACM Comput. Surv.* 46.4 (2013), 46:1–46:34. DOI: 10.1145/2528412.

[87] Marvin Hofer, Sebastian Hellmann, Milan Dojchinovski, and Johannes Frey. "The New DBpedia Release Cycle: Increasing Agility and Efficiency in Knowledge Extraction Workflows". In: *Semantic Systems. In the Era of Knowledge Graphs - 16th International Conference on Semantic Systems, SEMANTiCS 2020, Amsterdam, The Netherlands, September 7-10, 2020, Proceedings*, ed. by Eva Blomqvist et al. Vol. 12378. Lecture Notes in Computer Science. Springer, 2020, pp. 1–18. DOI: 10.1007/978-3-030-59833-4_1.

[88] Marvin Hofer, Daniel Obraczka, Alieh Saeedi, Hanna Köpcke, and Erhard Rahm. "Construction of Knowledge Graphs: State and Challenges". In: *CoRR* abs/2302.11509 (2023). DOI: 10.48550/arXiv.2302.11509. arXiv: 2302.11509.

[89] P. Holme and J. Saramäki. "Temporal networks". In: *Physics Reports* 519.3 (2012). Temporal Networks, pp. 97–125. ISSN: 0370-1573. DOI: https://doi.org/10.1016/j.physrep.2012.03.001.

[90] Petter Holme. "Temporal Networks". In: *Encyclopedia of Social Network Analysis and Mining, 2nd Edition*, ed. by Reda Alhajj et al. Springer, 2018. DOI: 10.1007/978-1-4939-7131-2_42.

[91] Jürgen Hölsch and Michael Grossniklaus. "An Algebra and Equivalences to Transform Graph Patterns in Neo4j". In: *Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference, EDBT/ICDT Workshops 2016, Bordeaux, France, March 15, 2016*, ed. by Themis Palpanas et al. Vol. 1558. CEUR Workshop Proceedings. CEUR-WS.org, 2016.

[92] Honeywell. *AEROSPACE & DEFENSE - Sensors and Switches Product Range Guide*. URL: https://prod-edam.honeywell.com/content/dam/honeywell-edam/sps/siot/ja/products/common/documents/sps-siot-aerospace-defense-rangeguide-000703-6-en-ciid-44438.pdf (visited on 08/01/2023).

[93] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. "PGX.D: A Fast Distributed Graph Processing Engine". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450337236. DOI: 10.1145/2807591.2807620.

[94] Pim van der Hoorn, Dong Yao, and Nelly Litvak. "Average nearest neighbor degrees in scale-free networks". In: *Internet Math.* 2018 (2018). DOI: 10.24166/IM.02.2018.

[95] 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 Conference on Information and Knowledge Management, CIKM 2016, Indianapolis, IN, USA, October 24-28, 2016*, ed. by Snehasis Mukhopadhyay et al. ACM, 2016, pp. 2469–2472. DOI: 10.1145/2983323.2983335.

[96] Shenyang Huang, Yasmeen Hitti, Guillaume Rabusseau, and Reihaneh Rabbany. "Laplacian Change Point Detection for Dynamic Graphs". In: *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, ed. by Rajesh Gupta et al. ACM, 2020, pp. 349–358. DOI: 10.1145/3394486.3403077.

[97] Alexandru Iosup et al. “LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms”. In: *Proc. VLDB Endow.* 9.13 (2016), pp. 1317–1328. DOI: 10.14778/3007263.3007270.

[98] ISO Central Secretary. *Information Technology - Database Languages - GQL*. en. Standard ISO/IEC WD 39075. Geneva, CH: International Organization for Standardization.

[99] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. “Time-evolving graph processing at scale”. In: *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, Redwood Shores, CA, USA, June 24 - 24, 2016*, ed. by Peter A. Boncz et al. ACM, 2016, p. 5. DOI: 10.1145/2960414.2960419.

[100] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E. Gonzalez, and Ion Stoica. “TEGRA: Efficient Ad-Hoc Analytics on Evolving Graphs”. In: *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, ed. by James Mickens et al. USENIX Association, 2021, pp. 337–355.

[101] Anuj Jain and Sartaj Sahni. “Foremost Walks and Paths in Interval Temporal Graphs”. In: *Algorithms* 15.10 (2022), p. 361. DOI: 10.3390/A15100361.

[102] Christian S. Jensen and Richard T. Snodgrass. “Temporal Data Management”. In: *IEEE Trans. Knowl. Data Eng.* 11.1 (1999), pp. 36–44. DOI: 10.1109/69.755613.

[103] H. Jeong, Z. Néda, and A. L. Barabási. “Measuring preferential attachment in evolving networks”. In: *Europhysics Letters* 61.4 (Feb. 2003), p. 567. DOI: 10.1209/epl/i2003-00166-9.

[104] Guodong Jin, Xiyang Feng, Ziyi Chen, Chang Liu, and Semih Salihoglu. “KÙZU Graph Database Management System”. In: *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org, 2023.

[105] Tom Johnston. *Bitemporal data: theory and practice*. Newnes, 2014.

[106] Martin Junghanns et al. “Analyzing Extended Property Graphs with Apache Flink”. In: *Proc. SIGMOD NDA Workshop*. 2016.

[107] Martin Junghanns et al. “GRADOOP: Scalable Graph Data Management and Analytics with Hadoop”. In: *CoRR* (2015).

[108] Martin Junghanns. *The Graph Definition Language (GDL)*. 2022. URL: https://github.com/dbs-leipzig/gdl (visited on 08/03/2023).

[109] 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, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017*, ed. by Peter A. Boncz et al. ACM, 2017, 3:1–3:8. DOI: 10.1145/3078447.3078450.

[110] 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*, ed. by Albert Y. Zomaya et al. Springer, 2017, pp. 457–505. DOI: 10.1007/978-3-319-49340-4_14.

[111] Martin Junghanns, André Petermann, and Erhard Rahm. "Distributed Grouping of Property Graphs with Gradoop". In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings*, ed. by Bernhard Mitschang et al. Vol. P-265. LNI. GI, 2017, pp. 103–122.

[112] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedhbi, Jeremy Chen, and Semih Salihoglu. "Graphflow: An Active Graph Database". In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, ed. by Semih Salihoglu et al. ACM, 2017, pp. 1695–1698. DOI: 10.1145/3035918.3056445.

[113] Udayan Khurana and Amol Deshpande. "Efficient snapshot retrieval over historical graph data". In: *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, ed. by Christian S. Jensen et al. IEEE Computer Society, 2013, pp. 997–1008. DOI: 10.1109/ICDE.2013.6544892.

[114] Hyoungshick Kim and Ross Anderson. "Temporal node centrality in complex networks". In: *Phys. Rev. E* 85 (2 Feb. 2012), p. 026107. DOI: 10.1103/PhysRevE.85.026107.

[115] Graham Klyne, Jeremy J Carroll, and Brian McBride. *Resource description framework (RDF): Concepts and abstract syntax*. 2018. URL: http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/ (visited on 08/10/2023).

[116] Ulrich Knauer and Kolja Knauer. *Morphisms, Monoids and Matrices*. Berlin, Boston: De Gruyter, 2019. ISBN: 9783110617368. DOI: doi:10.1515/9783110617368.

[117] Vassilis Kostakos. "Temporal graphs". In: *Physica A: Statistical Mechanics and its Applications* 388.6 (2009), pp. 1007–1023.

[118] Jürgen Krämer and Bernhard Seeger. "Semantics and implementation of continuous sliding window queries over data streams". In: *ACM Trans. Database Syst.* 34.1 (2009), 4:1–4:49. DOI: 10.1145/1508857.1508861.

[119] Matthias Kricke, Eric Peukert, and Erhard Rahm. “Graph Data Transformations in Gradoop”. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Proceedings*, ed. by Torsten Grust et al. Vol. P-289. LNI. Gesellschaft für Informatik, Bonn, 2019, pp. 193–202. DOI: 10.18420/BTW2019-12.

[120] Krishna G. Kulkarni and Jan-Eike Michels. “Temporal features in SQL: 2011”. In: *SIGMOD Rec.* 41.3 (2012), pp. 34–43. DOI: 10.1145/2380776.2380786.

[121] Yexin Li, Yu Zheng, Huichu Zhang, and Lei Chen. “Traffic prediction in a bike-sharing system”. In: *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, Bellevue, WA, USA, November 3-6, 2015*, ed. by Jie Bao et al. ACM, 2015, 33:1–33:10. DOI: 10.1145/2820783.2820837.

[122] Wouter Ligtenberg, Yulong Pei, George H. L. Fletcher, and Mykola Pechenizkiy. “Tink: A Temporal Graph Analytics Library for Apache Flink”. In: *Companion of the The Web Conference 2018 on The Web Conference 2018, WWW 2018, Lyon , France, April 23-27, 2018*, ed. by Pierre-Antoine Champin et al. ACM, 2018, pp. 71–72. DOI: 10.1145/3184558.3186934.

[123] Zheng Jye Ling, Quoc Trung Tran, Ju Fan, Gerald Choon Huat Koh, Thi Nguyen, Chuen Seng Tan, James Wei Luen Yip, and Meihui Zhang. “GEMINI: An Integrative Healthcare Analytics System”. In: *Proc. VLDB Endow.* 7.13 (2014), pp. 1766–1771. DOI: 10.14778/2733004.2733081.

[124] Xingyu Liu, Juan Chen, and Quan Wen. “A Survey on Graph Classification and Link Prediction based on GNN”. In: *CoRR* abs/2307.00865 (2023). DOI: 10.48550/ARXIV.2307.00865. arXiv: 2307.00865.

[125] Omar Lizardo and Isaac Jilbert. *Graph Metrics*. 2021. URL: http://olizardo.bol.ucla.edu/classes/soc-111/lessons-winter-2022/4-lesson-graph-metrics.html (visited on 08/10/2023).

[126] Google LLC. *Protocol Buffers*. 2023. URL: https://protobuf.dev (visited on 12/10/2023).

[127] Li Long, Khushnood Abbas, Niu Ling, and Syed Jafar Abbas. “Ranking Nodes in Temporal Networks: Eigen Value and Node Degree Growth based”. In: *2nd International Conference on Image Processing and Machine Vision*. 2020, pp. 146–153.

[128] Antonio Longa, Veronica Lachi, Gabriele Santin, Monica Bianchini, Bruno Lepri, Pietro Liò, Franco Scarselli, and Andrea Passerini. “Graph Neural Networks for temporal graphs: State of the art, open challenges, and opportunities”. In: *CoRR* abs/2302.01018 (2023). DOI: 10.48550/ARXIV.2302.01018. arXiv: 2302.01018.

[129] Lyft, Inc. *Citi Bike System Data*. 2023. URL: https://citibikenyc.com/system-data (visited on 08/01/2023).

[130] Patrick Mackey, Katherine Porterfield, Erin Fitzhenry, Sutanay Choudhury, and George Chin Jr. "A Chronological Edge-Driven Approach to Temporal Subgraph Isomorphism". In: *IEEE International Conference on Big Data (IEEE BigData 2018), Seattle, WA, USA, December 10-13, 2018*, ed. by Naoki Abe et al. IEEE, 2018, pp. 3972–3979. DOI: 10.1109/BIGDATA.2018.8622100.

[131] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: a system for large-scale graph processing". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, ed. by Ahmed K. Elmagarmid et al. ACM, 2010, pp. 135–146. DOI: 10.1145/1807167.1807184.

[132] Emaad A. Manzoor, Sadegh M. Milajerdi, and Leman Akoglu. "Fast Memory-efficient Anomaly Detection in Streaming Heterogeneous Graphs". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, ed. by Balaji Krishnapuram et al. ACM, 2016, pp. 1035–1044. DOI: 10.1145/2939672.2939783.

[133] Maria Massri, Zoltán Miklós, Philippe Raipin Parvédy, and Pierre Meye. "Clock-G: A temporal graph management system with space-efficient storage technique". In: *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 2022, pp. 2263–2276. DOI: 10.1109/ICDE53745.2022.00215.

[134] Andrew McGregor. "Graph stream algorithms: a survey". In: *SIGMOD Rec.* 43.1 (2014), pp. 9–20. DOI: 10.1145/2627692.2627694.

[135] Frank McSherry, Michael Isard, and Derek Gordon Murray. "Scalability! But at what COST?" In: *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*, ed. by George Candea. USENIX Association, 2015.

[136] Memgraph Ltd. *MemGraph - Open Source Graph Database*. 2023. URL: https://memgraph.com (visited on 08/25/2023).

[137] Pasquale De Meo, Emilio Ferrara, Giacomo Fiumara, and Alessandro Provetti. "Generalized Louvain method for community detection in large networks". In: *11th International Conference on Intelligent Systems Design and Applications, ISDA 2011, Córdoba, Spain, November 22-24, 2011*, ed. by Sebastián Ventura et al. IEEE, 2011, pp. 88–93. DOI: 10.1109/ISDA.2011.6121636.

[138] 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". In: *ACM Trans. Storage* 11.3 (2015), 14:1–14:34. DOI: 10.1145/2700302.

[139] Vera Zaychik Moffitt and Julia Stoyanovich. "Temporal graph algebra". In: *Proceedings of The 16th International Symposium on Database Programming Languages, DBPL 2017, Munich, Germany, September 1, 2017*, ed. by Tiark Rompf et al. ACM, 2017, 10:1–10:12. DOI: 10.1145/3122831.3122838.

[140] Neo4j. *GitHub Repository - Morpheus: Cypher for Apache Spark*. 2023. URL: https://github.com/opencypher/morpheus (visited on 08/10/2023).

[141] Neo4j. *Kafka Connect Neo4j Connector User Guide*. 2023. URL: https://neo4j.com/docs/kafka/ (visited on 08/10/2023).

[142] Neo4j. *The Neo4j Graph Platform*. 2018. URL: https://neo4j.com/ (visited on 08/03/2023).

[143] Neo4j Inc. *APOC Documentation - Graph Grouping*. URL: https://neo4j.com/labs/apoc/4.1/virtual/graph-grouping/ (visited on 08/03/2023).

[144] M. Newman. "Clustering and preferential attachment in growing networks". In: *Physical review. E, Statistical, nonlinear, and soft matter physics* 64 (Sept. 2001), p. 025102. DOI: 10.1103/PhysRevE.64.025102.

[145] Vincenzo Nicosia, John Kit Tang, Cecilia Mascolo, Mirco Musolesi, Giovanni Russo, and Vito Latora. "Graph Metrics for Temporal Networks". In: vol. abs/1306.0493. 2013. arXiv: 1306.0493.

[146] Mikael Nilsson, Jonas Kvarnström, and Patrick Doherty. "Efficient processing of simple temporal networks with uncertainty: algorithms for dynamic controllability verification". In: *Acta Informatica* 53.6-8 (2016), pp. 723–752. DOI: 10.1007/S00236-015-0248-8.

[147] Daniel Obraczka, Alieh Saeedi, and Erhard Rahm. "Knowledge Graph Completion with FAMER (DI2KG Challenge Winner)". In: *Proceedings of the 1st International Workshop on Challenges and Experiences from Data Integration to Knowledge Graphs co-located with the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD 2019), Anchorage, Alaska, August 5, 2019*, ed. by Donatella Firmani et al. Vol. 2512. CEUR Workshop Proceedings. CEUR-WS.org, 2019.

[148] openCypher. *Cypher Query Language Reference*. Feb. 2018. URL: https://github.com/opencypher/openCypher/blob/master/docs/openCypher9.pdf (visited on 08/03/2023).

[149] Oracle. *Continuous Query Notification (CQN)*. URL: https://cx-oracle.readthedocs.io/en/latest/user%5C_guide/cqn.html (visited on 08/03/2023).

[150] Oracle. *PGQL - Property Graph Query Language*. 2023. URL: https://pgql-lang.org/ (visited on 08/03/2023).

[151] Oracle. *PGQL 1.3 Specification*. 2020. URL: https://pgql-lang.org/spec/1.3/ (visited on 08/03/2023).

[152] *OrientDB Community*. 2020. URL: http://www.orientechnologies.com/orientdb/ (visited on 08/01/2023).

[153] Diego Orlando, Joaquin Ormachea, Valeria Soliani, and Alejandro Ariel Vaisman. "TGV: A Visualization Tool for Temporal Property Graph Databases". In: *Information Systems Frontiers* (2023), pp. 1–22.

[154] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, 4th Edition*. Springer, 2020. ISBN: 978-3-030-26252-5. DOI: 10.1007/978-3-030-26253-2.

[155] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. "Evaluating Complex Queries on Streaming Graphs". In: *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 2022, pp. 272–285. DOI: 10.1109/ICDE53745.2022.00025.

[156] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. "Regular Path Query Evaluation on Streaming Graphs". In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, ed. by David Maier et al. ACM, 2020, pp. 1415–1430. DOI: 10.1145/3318464.3389733.

[157] Georgios A. Pavlopoulos, Maria Secrier, Charalampos N. Moschopoulos, Theodoros G. Soldatos, Sophia Kossida, Jan Aerts, Reinhard Schneider, and Pantelis G. Bagos. "Using graph theory to analyze biological networks". In: *BioData Min.* 4 (2011), p. 10. DOI: 10.1186/1756-0381-4-10.

[158] André Petermann. "On Pattern Mining in Graph Data to Support Decision-Making". PhD thesis. Leipzig University, Germany, 2019.

[159] Andre Petermann, Martin Junghanns, Robert Müller, and Erhard Rahm. "BIIIG: Enabling business intelligence with integrated instance graphs". In: *Workshops Proceedings of the 30th International Conference on Data Engineering Workshops, ICDE 2014, Chicago, IL, USA, March 31 - April 4, 2014*. IEEE Computer Society, 2014, pp. 4–11. DOI: 10.1109/ICDEW.2014.6818294.

[160] 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, BDCAT 2017, Austin, TX, USA, December 05 - 08, 2017*, ed. by Ashiq Anjum et al. ACM, 2017, pp. 237–246. DOI: 10.1145/3148055.3148064.

[161] Evaggelia Pitoura. "Historical Graphs: Models, Storage, Processing". In: *Business Intelligence and Big Data - 7th European Summer School, eBISS 2017, Bruxelles, Belgium, July 2-7, 2017, Tutorial Lectures*, ed. by Esteban Zimányi. Vol. 324. Lecture Notes in Business Information Processing. Springer, 2017, pp. 84–111. DOI: 10.1007/978-3-319-96655-7_4.

[162] Tahereh Pourhabibi, Kok-Leong Ong, Booi H. Kam, and Yee Ling Boo. "Fraud detection: A systematic literature review of graph-based anomaly detection approaches". In: *Decision Support Systems* 133 (2020), p. 113303. ISSN: 0167-9236. DOI: https://doi.org/10.1016/j.dss.2020.113303.

[163] Shipeng Qi et al. "The LDBC Financial Benchmark". In: *CoRR* abs/2306.15975 (2023). DOI: 10.48550/ARXIV.2306.15975. arXiv: 2306.15975.

[164] Pablo Jensen Remy Cazabet and Pierre Borgnat. "Tracking the evolution of temporal patterns of usage in bicycle-Sharing systems using nonnegative matrix factorization on multiple sliding windows". In: *International Journal of Urban Sciences* 22.2 (2018), pp. 147–161. DOI: 10.1080/12265934.2017.1336468. eprint: https://doi.org/10.1080/12265934.2017.1336468.

[165] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. "On Querying Historical Evolving Graph Sequences". In: *Proc. VLDB Endow.* 4.11 (2011), pp. 726–737.

[166] 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, Redwood Shores, CA, USA, June 24 - 24, 2016*, ed. by Peter A. Boncz et al. ACM, 2016, p. 7. DOI: 10.1145/2960414.2960421.

[167] Riccardo Tommasini and Christopher Rost and Julian Pielmaier and Christopher Lausch. *Seraph RSP4J implementation GitHub*. 2023. URL: https://github.com/dbs-leipzig/seraph-rsp4j-impl (visited on 08/10/2023).

[168] Christopher S Riolo, James S Koopman, and Stephen E Chick. "Methods and measures for the description of epidemiologic contact networks". In: *J Urban Health* 78.3 (2001), pp. 446–457.

[169] Marko A. Rodriguez. “The Gremlin graph traversal machine and language (invited talk)”. In: *Proceedings of the 15th Symposium on Database Programming Languages, Pittsburgh, PA, USA, October 25-30, 2015*, ed. by James Cheney et al. ACM, 2015, pp. 1–10. DOI: 10.1145/2815072.2815073.

[170] Marko A. Rodriguez and Peter Neubauer. “Constructions from dots and lines”. In: *Bulletin of the American Society for Information Science and Technology* 36.6 (2010), pp. 35–41. DOI: https://doi.org/10.1002/bult.2010.1720360610. eprint: https://asistdl.onlinelibrary.wiley.com/doi/pdf/10.1002/bult.2010.1720360610.

[171] Giulio Rossetti and Rémy Cazabet. “Community Discovery in Dynamic Networks: A Survey”. In: *ACM Comput. Surv.* 51.2 (2018), 35:1–35:37. DOI: 10.1145/3172867.

[172] **Christopher Rost**, Philip Fritzsche, Kevin Gomez, Timo Adameit, and Lucas Schons. *GitHub Wiki of Gradoop*. 2020. URL: https://github.com/dbs-leipzig/gradoop/wiki (visited on 10/10/2022).

[173] **Christopher Rost**, Philip Fritzsche, Lucas Schons, Maximilian Zimmer, Dieter Gawlick, and Erhard Rahm. “Bitemporal Property Graphs to Organize Evolving Systems”. In: *CoRR* abs/2111.13499 (2021). arXiv: 2111.13499.

[174] **Christopher Rost**, Kevin Gómez, Peter Christen, and Erhard Rahm. “Evolution of Degree Metrics in Large Temporal Graphs”. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2023), 20. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 06.-10, März 2023, Dresden, Germany, Proceedings*, ed. by Birgitta König-Ries et al. Vol. P-331. LNI. Gesellschaft für Informatik e.V., 2023, pp. 485–507. ISBN: 978-3-88579-725-8. DOI: 10.18420/BTW2023-23.

[175] **Christopher Rost**, Kevin Gómez, Philip Fritzsche, Andreas Thor, and Erhard Rahm. “Exploration and Analysis of Temporal Property Graphs”. In: *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021*, ed. by Yannis Velegrakis et al. OpenProceedings.org, 2021, pp. 682–685. ISBN: 978-3-89318-084-4. DOI: 10.5441/002/EDBT.2021.83.

[176] **Christopher Rost**, Kevin Gómez, Matthias Täschner, Philip Fritzsche, Lucas Schons, Lukas Christ, Timo Adameit, Martin Junghanns, and Erhard Rahm. “Distributed temporal graph analytics with Gradoop”. In: *VLDB J.* 31.2 (2022), pp. 375–401. DOI: 10.1007/S00778-021-00667-4.

[177] **Christopher Rost**, Andreas Thor, Philip Fritzsche, Kevin Gómez, and Erhard Rahm. “Evolution Analysis of Large Graphs with Gradoop”. In: *Machine Learning and Knowledge Discovery in Databases - International Workshops of ECML PKDD 2019, Würzburg, Germany, September 16-20, 2019, Proceedings, Part I*, ed. by Peggy

Cellier et al. Vol. 1167. Communications in Computer and Information Science. Springer, 2019, pp. 402–408. ISBN: 978-3-030-43822-7. DOI: 10.1007/978-3-030-43823-4_33.

[178] **Christopher Rost**, Andreas Thor, and Erhard Rahm. "Analyzing Temporal Graphs with Gradoop". In: *Datenbank-Spektrum* 19.3 (2019), pp. 199–208. DOI: 10.1007/S13222-019-00325-8.

[179] **Christopher Rost**, Andreas Thor, and Erhard Rahm. "Temporal Graph Analysis using Gradoop". In: *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Workshopband*, ed. by Holger Meyer et al. Vol. P-290. LNI. Gesellschaft für Informatik, Bonn, 2019, pp. 109–118. ISBN: 978-3-88579-684-8. DOI: 10.18420/BTW2019-WS-11.

[180] **Christopher Rost**, Riccardo Tommasini, Angela Bonifati, Emanuele Della Valle, Erhard Rahm, Keith W. Hare, Stefan Plantikow, Petra Selmer, and Hannes Voigt. "Seraph: Continuous Queries on Property Graph Streams". In: *Proceedings of the 27td International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy, March 25 - March 28, 2024*, ed. by Letizia Tanca et al. OpenProceedings.org, 2024. DOI: 10.48786/edbt.2024.21.

[181] M. Ali Rostami, Matthias Kricke, Eric Peukert, Stefan Kühne, Moritz Wilke, Steffen Dienst, and Erhard Rahm. "BIGGR: Bringing Gradoop to Applications". In: *Datenbank-Spektrum* 19 (2019), pp. 51–60. DOI: 10.1007/s13222-019-00306-x.

[182] M. Ali Rostami, Eric Peukert, Moritz Wilke, and Erhard Rahm. "Big graph analysis by visually created workflows". In: *BTW 2019*, ed. by Torsten Grust et al. 2019, pp. 559–563. DOI: 10.18420/btw2019-45.

[183] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. "SynopSys: large graph analytics in the SAP HANA database through summarization". In: *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-located with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013*, ed. by Peter A. Boncz et al. CWI/ACM, 2013, p. 16. DOI: 10.1145/2484425.2484441.

[184] Alieh Saeedi, Markus Nentwig, Eric Peukert, and Erhard Rahm. "Scalable Matching and Clustering of Entities with FAMER". In: *Complex Syst. Informatics Model. Q.* 16 (2018), pp. 61–83. DOI: 10.7250/CSIMQ.2018-16.04.

[185] Alieh Saeedi, Eric Peukert, and Erhard Rahm. "Incremental Multi-source Entity Resolution for Knowledge Graph Completion". In: *The Semantic Web - 17th International Conference, ESWC 2020, Heraklion, Crete, Greece, May 31-June 4, 2020,*

Proceedings, ed. by Andreas Harth et al. Vol. 12123. Lecture Notes in Computer Science. Springer, 2020, pp. 393–408. DOI: 10.1007/978-3-030-49461-2_23.

[186] Alieh Saeedi, Eric Peukert, and Erhard Rahm. "Using Link Features for Entity Clustering in Knowledge Graphs". In: *The Semantic Web - 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3-7, 2018, Proceedings*, ed. by Aldo Gangemi et al. Vol. 10843. Lecture Notes in Computer Science. Springer, 2018, pp. 576–592. DOI: 10.1007/978-3-319-93417-4_37.

[187] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. "The ubiquity of large graphs and surprising challenges of graph processing: extended survey". In: *VLDB J.* 29.2-3 (2020), pp. 595–618. DOI: 10.1007/S00778-019-00548-X.

[188] Sherif Sakr and Ghazi Al-Naymat. "Graph indexing and querying: a review". In: *Int. J. Web Inf. Syst.* 6.2 (2010), pp. 101–120. DOI: 10.1108/17440081011053104.

[189] Sherif Sakr et al. "The future is big graphs: a community view on graph processing systems". In: *Commun. ACM* 64.9 (2021), pp. 62–71. DOI: 10.1145/3434642.

[190] Jari Saramäki and Kimmo Kaski. "Modelling development of epidemics with dynamic small-world networks". In: *Journal of Theoretical Biology* 234.3 (2005), pp. 413–421. ISSN: 0022-5193. DOI: https://doi.org/10.1016/j.jtbi.2004.12.003.

[191] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. "The Graph Neural Network Model". In: *IEEE Trans. Neural Networks* 20.1 (2009), pp. 61–80. DOI: 10.1109/TNN.2008.2005605.

[192] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. "The Hadoop Distributed File System". In: *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, ed. by Mohammed G. Khatib et al. IEEE Computer Society, 2010, pp. 1–10. DOI: 10.1109/MSST.2010.5496972.

[193] Keith Smith and Javier Escudero. "Normalised degree variance". In: *Appl. Netw. Sci.* 5.1 (2020), p. 32. DOI: 10.1007/S41109-020-00273-3.

[194] Tom AB Snijders. "The degree variance: an index of graph heterogeneity". In: *Social networks* 3.3 (1981), pp. 163–174. DOI: 10.1016/0378-8733(81)90014-9.

[195] Caroline Spivack. *Halloween Dog Parade 2018: What You Need To Know*. 2018. URL: https://patch.com/new-york/east-village/halloween-dog-parade-2018-what-you-need-know (visited on 10/10/2023).

[196] Stack Exchange, Inc. *Stack Exchange Data Dump*. 2021. URL: https://archive.org/details/stackexchange (visited on 08/03/2023).

[197] Benjamin A. Steer, Félix Cuadrado, and Richard G. Clegg. "Raphtory: Streaming analysis of distributed temporal graphs". In: *Future Gener. Comput. Syst.* 102 (2020), pp. 453–464. DOI: 10.1016/J.FUTURE.2019.08.022.

[198] Michael Stonebraker, Ugur Çetintemel, and Stanley B. Zdonik. "The 8 requirements of real-time stream processing". In: *SIGMOD Rec.* 34.4 (2005), pp. 42–47. DOI: 10.1145/1107499.1107504.

[199] Jiao Tao, Evren Sirin, Jie Bao, and Deborah L. McGuinness. "Integrity Constraints in OWL". In: *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, ed. by Maria Fox et al. AAAI Press, 2010, pp. 1443–1448. DOI: 10.1609/aaai.v24i1.7525.

[200] Robert Endre Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM J. Comput.* 1.2 (1972), pp. 146–160. DOI: 10.1137/0201010.

[201] Douglas B. Terry, David Goldberg, David A. Nichols, and Brian M. Oki. "Continuous Queries over Append-Only Databases". In: *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 2-5, 1992*, ed. by Michael Stonebraker. ACM Press, 1992, pp. 321–330. DOI: 10.1145/130283.130333.

[202] The Apache Software Foundation. *Apache Flink - DataSet API*. 2023. URL: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/dev/dataset/overview/ (visited on 08/10/2023).

[203] *The Banks Association of Turkey: Statistical Report*. URL: http://www.tbb.org.tr/en/banks-and-banking-sector-information/statistical-reports/20 (visited on 08/01/2023).

[204] William Hedley Thompson, Per Brantefors, and Peter Fransson. "From static to temporal network theory: Applications to functional brain connectivity". In: *Network Neuroscience* 1.2 (2017), pp. 69–99.

[205] Inc. TigerGraph. *TigerGraph graph database*. 2022. URL: https://www.tigergraph.com (visited on 10/10/2022).

[206] Aizhan Tlebaldinova, Aliya Nugumanova, Yerzhan Baiburin, Zheniskul Zhantassova, Markhaba Karmenova, and Andrey Ivanov. "Temporal Network Approach to Explore Bike Sharing Usage Patterns". In: *Proceedings of the 6th International Conference on Vehicle Technology and Intelligent Transport Systems, VEHITS 2020, Prague, Czech Republic, May 2-4, 2020*, ed. by Karsten Berns et al. SCITEPRESS, 2020, pp. 129–136. DOI: 10.5220/0009575901290136.

[207] Neo4j Inc. Tom Geudens. *POLE Investigations*. 2022. URL: https://www.slideshare.net/neo4j/pole-investigations (visited on 10/10/2022).

[208] Riccardo Tommasini, Pieter Bonte, Femke Ongenae, and Emanuele Della Valle. "RSP4J: An API for RDF Stream Processing". In: *The Semantic Web - 18th International Conference, ESWC 2021, Virtual Event, June 6-10, 2021, Proceedings*, ed. by Ruben Verborgh et al. Vol. 12731. Lecture Notes in Computer Science. Springer, 2021, pp. 565–581. DOI: 10.1007/978-3-030-77385-4_34.

[209] Riccardo Tommasini, Pieter Bonte, Fabiano Spiga, and Emanuele Della Valle. *Streaming linked data : from vision to practice*. eng. Springer, 2023, XVI, 158. ISBN: 9783031153709. DOI: 10.1007/978-3-031-15371-6.

[210] Riccardo Tommasini, Sherif Sakr, Emanuele Della Valle, and Hojjat Jafarpour. "Declarative Languages for Big Streaming Data". In: *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, ed. by Angela Bonifati et al. OpenProceedings.org, 2020, pp. 643–646. DOI: 10.5441/002/edbt.2020.84.

[211] Jonas Traub, Philipp Marian Grulich, Alejandro Rodriguez Cuellar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. "Scotty: General and Efficient Open-source Window Aggregation for Stream Processing Systems". In: *ACM Trans. Database Syst.* 46.1 (2021), 1:1–1:46. DOI: 10.1145/3433675.

[212] Leslie G. Valiant. "A Bridging Model for Parallel Computation". In: *Commun. ACM* 33.8 (1990), pp. 103–111. DOI: 10.1145/79173.79181.

[213] Juliane Verwiebe, Philipp M. Grulich, Jonas Traub, and Volker Markl. "Survey of window types for aggregation in stream processing systems". In: *VLDB J.* 32.5 (2023), pp. 985–1011. DOI: 10.1007/s00778-022-00778-6.

[214] Sofya Vorotnikova. "Massive Graph Analysis in the Data Stream Model". In: (2019).

[215] Yishu Wang, Ye Yuan, Yuliang Ma, and Guoren Wang. "Time-Dependent Graphs: Definitions, Applications, and Algorithms". In: *Data Sci. Eng.* 4.4 (2019), pp. 352–366. DOI: 10.1007/S41019-019-00105-0.

[216] Zhengkui Wang, Qi Fan, Huiju Wang, Kian-Lee Tan, Divyakant Agrawal, and Amr El Abbadi. "Pagrol: Parallel graph olap over large-scale attributed graphs". In: *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, ed. by Isabel F. Cruz et al. IEEE Computer Society, 2014, pp. 496–507. DOI: 10.1109/ICDE.2014.6816676.

[217] Zhiqiang Wang, Xubin Pei, Yanbo Wang, and Yiyang Yao. "Ranking the key nodes with temporal degree deviation centrality on complex networks". In: *2017 29th Chinese Control And Decision Conference (CCDC)*. IEEE. 2017, pp. 1484–1489. DOI: 10.1109/CCDC.2017.7978752.

[218] Jack Waudby, Benjamin A. Steer, Arnau Prat-Pérez, and Gábor Szárnyas. "Supporting Dynamic Graphs and Temporal Entity Deletions in the LDBC Social Network Benchmark's Data Generator". In: *GRADES-NDA'20: Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Portland, OR, USA, June 14, 2020*, ed. by Akhil Arora et al. ACM, 2020, 8:1–8:8. DOI: 10.1145/3398682.3399165.

[219] Jim Webber. *White paper: The Top 10 Use Cases for Graph Technology*. 2020. URL: https://neo4j.com/whitepapers/top-ten-use-cases-graph-database-technology/ (visited on 10/01/2023).

[220] Daniel ten Wolde, Gábor Szárnyas, and Peter A. Boncz. "DuckPGQ: Bringing SQL/PGQ to DuckDB". In: *Proc. VLDB Endow.* 16.12 (2023), pp. 4034–4037. DOI: 10.14778/3611540.3611614.

[221] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. "Path Problems in Temporal Graphs". In: *Proc. VLDB Endow.* 7.9 (2014), pp. 721–732. DOI: 10.14778/2732939.2732945.

[222] Shunxin Xiao, Shiping Wang, Yuanfei Dai, and Wenzhong Guo. "Graph neural networks in node classification: survey and evaluation". In: *Mach. Vis. Appl.* 33.1 (2022), p. 4. DOI: 10.1007/S00138-021-01251-0.

[223] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. "GraphX: a resilient distributed graph system on Spark". In: *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-located with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013*, ed. by Peter A. Boncz et al. CWI/ACM, 2013, p. 2. DOI: 10.1145/2484425.2484427.

[224] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, ed. by Steven D. Gribble et al. USENIX Association, 2012, pp. 15–28.

[225] Matei Zaharia et al. "Apache Spark: a unified engine for big data processing". In: *Commun. ACM* 59.11 (2016), pp. 56–65. DOI: 10.1145/2934664.

[226] Aya Zaki, Mahmoud Attia, Doaa Hegazy, and Safaa Amin. "Comprehensive survey on dynamic graph models". In: *International Journal of Advanced Computer Science and Applications* 7.2 (2016).

[227] Maximilian Zimmer. "Kontinuierliche Graph-Query Notifikationen auf Basis eines RDBMS". In: *Bachelor thesis at Universität Leipzig* (2021).

Selbständigkeitserklärung

Hiermit erkläre ich, die vorliegende Dissertation selbständig und ohne unzulässige fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angeführten Quellen und Hilfsmittel benutzt und sämtliche Textstellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Schriften entnommen wurden, und alle Angaben, die auf mündlichen Auskünften beruhen, als solche kenntlich gemacht. Ebenfalls sind alle von anderen Personen bereitgestellten Materialien oder erbrachten Dienstleistungen als solche gekennzeichnet.

Leipzig, 18. Dezember 2023 Christopher Rost