The VLDB Journal (2022) 31:375–401
https://doi.org/10.1007/s00778-021-00667-4

SPECIAL ISSUE PAPER

Distributed temporal graph analytics with GRADOOP

Christopher Rost[1] · Kevin Gomez[1] · Matthias Täschner[1] · Philip Fritzsche[1] · Lucas Schons[1] · Lukas Christ[1] · Timo Adameit[1] · Martin Junghanns[2] · Erhard Rahm[1]

Received: 14 September 2020 / Revised: 18 February 2021 / Accepted: 26 March 2021 / Published online: 19 May 2021

Abstract
Temporal property graphs are graphs whose structure and properties change over time. Temporal graph datasets tend to be large due to stored historical information, asking for scalable analysis capabilities. We give a complete overview of GRADOOP, a graph dataflow system for scalable, distributed analytics of temporal property graphs which has been continuously developed since 2005. Its graph model TPGM allows bitemporal modeling not only of vertices and edges but also of graph collections. A declarative analytical language called GRALA allows analysts to flexibly define analytical graph workflows by composing different operators that support temporal graph analysis. Built on a distributed dataflow system, large temporal graphs can be processed on a shared-nothing cluster. We present the system architecture of GRADOOP, its data model TPGM with composable temporal graph operators, like snapshot, difference, pattern matching, graph grouping and several implementation details. We evaluate the performance and scalability of selected operators and a composed workflow for synthetic and real-world temporal graphs with up to 283 M vertices and 1.8 B edges, and a graph lifetime of about 8 years with up to 20 M new edges per year. We also reflect on lessons learned from the GRADOOP effort.

Keywords Graph processing · Temporal graph · Distributed processing · Graph analytics · Bitemporal graph model

1 Introduction

Graphs are simple, yet powerful data structures to model and analyze relations between real-world data objects. The analysis of graph data has recently gained increasing interest, e.g., for web information systems, social networks [23], business intelligence [60,69,81] or in life science applications [20,51]. A powerful class of graphs are so-called knowledge graphs, such as in the current CovidGraph project[1], to semantically represent heterogeneous entities (gene mutations, publications, patients, etc.) and their relations from different data sources, e.g., to provide consolidated and integrated knowledge to improve data analysis. There is a large spectrum of analysis forms for graph data, ranging from graph queries to find certain 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. Graphs 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 [42,73]. Furthermore, the structure and contents of graphs and networks mostly change over time

✉ Christopher Rost
rost@informatik.uni-leipzig.de

Kevin Gomez
gomez@informatik.uni-leipzig.de

Matthias Täschner
taeschner@informatik.uni-leipzig.de

Philip Fritzsche
fritzsche@informatik.uni-leipzig.de

Lucas Schons
schons@informatik.uni-leipzig.de

Lukas Christ
christ@informatik.uni-leipzig.de

Timo Adameit
adameit@informatik.uni-leipzig.de

Martin Junghanns
martin.junghanns@neo4j.com

Erhard Rahm
rahm@informatik.uni-leipzig.de

[1] University of Leipzig & ScaDS.AI Dresden/Leipzig, Leipzig, Germany

[2] Neo4j, Inc., San Mateo, USA

[1] https://covidgraph.org/.

making it necessary to continuously evolve the graph data and 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 which people have been in the same department for a certain duration of time or to find out how collaborations between universities or friendships in a social network evolve, how an infectious disease spreads over time, etc. Like in bitemporal databases [37,38], 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. This management of graph data along two timelines allows one to keep a complete history of the past database states as well as to track the provenance of information with full governance and immutability. It also allows to query across both valid and system time axes, e.g., to answer questions like "What friends did Alice have on last August 1st as we knew it on September 1st?".

Two major categories of systems focus on the management and analysis of graph data: graph database systems and distributed graph processing systems [42]. A closer look at both categories and their strengths and weaknesses is given in Sect 2. So graph database systems are typically less suited for high-volume data analysis and graph mining [31,53,75] as they often do not support distributed processing on partitioned graphs which limits the maximum graph size and graph analysis to the resources of a single machine. By contrast, distributed graph processing systems support high scalability and parallel graph processing but typically lack an expressive graph data model and declarative query support [13,42]. In particular, the latter makes it difficult for users to formulate complex analytical tasks as this requires profound programming and system knowledge. While a single graph can be processed and modeled, the support for storing and analyzing many distinct graphs is usually missing in these systems. Further, both categories have typically neither native support for a temporal graph data model [49,74] nor for temporal graph analysis and querying.

To overcome the limitations of these system approaches and to combine their strengths, we started in 2015 already the development of a new open-source[2] distributed graph analysis platform called GRADOOP (**Gra**ph Analytics on Ha**doop**) that has continuously been extended in the last years [29,39–41,43,65,66]. GRADOOP is a distributed platform to achieve high scalability and parallel graph processing. It is based on an extended property graph model supporting the processing both 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 the creation and analysis of static graphs, we have recently added support for bitemporal graphs making GRADOOP a distributed platform for temporal graph analysis.

In this work, we present a complete system overview of GRADOOP with a focus on the latest extensions for temporal property graphs. This addition required adjustments in all components of the system, as well as 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. We also reflect on lessons learnt from the GRADOOP effort.

The main contributions are thus 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.
- *Lessons learned* We briefly summarize findings from five years of research on distributed graph analysis with GRADOOP.

After an overview of the current graph system landscape (Sect. 2), the architecture of the GRADOOP framework is described in Sect. 3. The bitemporal graph data model including an outline of all available operators and a detailed description of selected ones is given in Sect. 4. After explaining implementation details (Sect. 5), selected operators are evaluated in Sect 6. In Sect. 7, we discuss lessons learned, related projects and ongoing work. We summarize our work in Sect. 8.

[2] https://github.com/dbs-leipzig/gradoop.

2 Graph system landscape

GRADOOP relates to graph data processing and temporal databases, two areas with a huge amount of previous research. We will thus mostly focus on the discussion of temporal extensions of property graphs and its query languages and their support within graph databases and distributed graph processing systems. We also point out how GRADOOP differs from previous approaches.

2.1 Temporal property graphs and query languages

A property graph [11,63] is a directed graph where vertices and edges can have an arbitrary number of *properties* that are typically represented as key-value pairs. *Temporal property graphs* are property graphs that reflect the graph's evolution, i.e., changes of the graph structure and of properties over time. There are several surveys about such temporal graphs [18,32,45] which have also been called temporal networks [32], time-varying graphs [18], time-dependent graphs [80], evolving graphs and other names [27,55,66,77,82].

Temporal models are quite mature for relational databases and support for bitemporal tables and time-related queries have been incorporated into SQL:2011 [37,38,47]. By contrast, temporal *graph* models still differ in many aspects so that there is not yet a consensus about the most promising approach (e.g., [18,27,55,77]). Such differences exist regarding the supported time dimensions (valid time, transaction time or both/bitemporal), the kinds of possible changes on the graph structure and of properties, and whether a temporal graph is represented as a series of graph snapshots or as a single graph, e.g., reflecting changes by time properties.

A temporal property graph model should also include a query language that utilizes the temporal information, e.g., to analyze different states or the evolution of the graph data. Current declarative query languages for property graph databases, such as Cypher [26,58], Gremlin [62] or Oracle PGQL [79], are powerful languages supporting mining for complex graph patterns (pattern matching), navigational expressions or aggregation queries [10,12]. However, they assume static property graphs and have no built-in support for temporal queries that goes beyond the use of time or date properties. In addition, special interval types are missing to represent a period in time with concrete start and end timestamps, and relations between such time intervals, e.g., as defined by Allen [9]. 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.

GRADOOP provides a simple yet powerful bitemporal property graph model TPGM and temporal query support that avoids the mentioned limitations (see Sect 4). The model supports the processing of not only single property graphs but also of collections of such graphs. Temporal information for valid and transaction time is represented within specific attributes, thereby avoiding the dedicated storage of graph snapshots (snapshots can still be determined). The processing of temporal graphs is supported by temporal graph operators that can be composed within analytical programs. We also support temporal queries based on *TemporalGDL*, a Cypher-like pattern matching language combined with extensions adapted from SQL:2011 [47] and Allen's interval algebra [9] that can use diverse temporal patterns in a declarative manner (see Sect 4.2).

2.2 Graph database and graph processing systems

Graph database systems are typically based on the property graph model (PGM) (or RDF [44]) and support a query language supporting operations such as pattern matching [10] and neighborhood traversal. The analysis of current graph database systems in [42] showed that they mostly 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 certain 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 now also support partitioned graph storage). As already discussed for the query languages, the focus is on static graphs so that the storage and analysis of (bi)temporal graphs are not supported.

Graph processing systems are typically based on the bulk synchronous parallel (BSP) [78] programming model and provide scalability, fault tolerance and flexibility to express arbitrary static graph algorithms. The analysis in [42] 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 that the management and use of temporal information is left to the applications.

GRADOOP aims at combining the advantages of graph database and graph processing systems and to additionally provide several extensions, in particular support for bitemporal graphs and temporal graph analysis. As already 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 so that parallel graph analysis on distributed cluster platforms is supported for horizontal scalability.

2.3 Temporal graph processing systems

We now discuss systems for temporal graph processing that have been developed in the last decade. Kineograph [21] 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 [54] (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 [25] 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 [77] 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 [36] 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 predetermined temporal queries and workflows.

Tink [50] is a library for analyzing temporal property graphs built on Apache Flink. Temporal information is represented by single time intervals for edges only, i.e., there is no temporal information for vertices and no support for bitemporality. It focuses on temporal path problems and the calculation of graph measures such as temporal betweenness and closeness. The systems TGraph [34] and Graphite [27] 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 [6] to reduce the temporal resolution for explorative graph analysis, similar to GRADOOP's grouping operator (see Sect. 4.2).

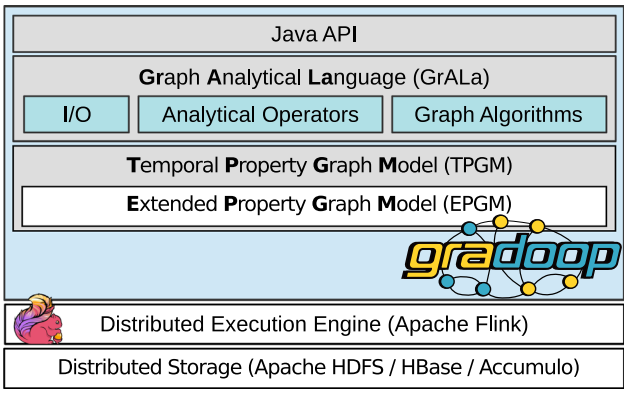

Fig. 1 Gradoop high-level architecture

Compared to these systems, 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 a less complete functionality regarding declarative graph operators and the possibility to combine them in 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.

3 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 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.

Table 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)

	Analytical Operators		Graph Algorithms
Temporal Graph	Aggregation	Combination	PageRank
	Pattern Matching [41]	Overlap	Community Detection
	Transformation	Exclusion	Connected Components
	Grouping [43,66]	Equality	Single Source Shortest Path
	Subgraph	*Call**	Summarization
	Snapshot [66]		Hyperlink-Induced Topic Search
	Difference [66]		
Graph Coll.	Selection	Difference	Frequent Subgraph Mining [61]
	Distinct	Equality	
	Limit	*Apply**	
	Union	*Reduce**	
	Intersection	*Call**	

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 reading and writing graphs from the local file system or the Apache Hadoop Distributed File System (HDFS) [76]. GRADOOP in addition supports Apache HBase [2] and Apache Accumulo [1] 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. 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 Sect. 5.5.

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 is 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 [8,16]. 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 5.1.

Temporal property graph model The TPGM [66] describes how graphs and their evolution are represented in GRADOOP. It is an extension of the extended property graph model (EPGM) [40] which is based on the widely accepted property graph model [11,63]. To handle the evolution of the graph and its elements (vertices and edges), the model uses concepts of bitemporal data management [37,38] 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 4.1 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 1 shows a subset of frequently used analytical operators and graph algorithms categorized by their input [40,66]. 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* [41] and *graph grouping* [43,66]. Furthermore, there are dedicated transformation operators to support data integration [46]. 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 Sect. 4.2.

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 [14]. This extension makes it possible to use selected GRALA operators within KNIME analysis workflows and to execute the resulting workflows on a remote cluster [67,68]. 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 Sect. 4.2.6). Furthermore, we provide extensive support for entity resolution and entity clustering within a dedicated framework called FAMER [70–72] 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 [59].

4 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. 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.

4.1 Graph data model

The Property Graph Model (PGM) [11,63] is a widely accepted graph data model used by many graph database systems [10], e.g., JanusGraph [4], OrientDB [5], Oracle's Graph Database [22] and Neo4j [57]. 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 [38]. In addition, the TPGM supports graph collections, which were introduced by the EPGM [40], 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[3] 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 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:

[3] The preliminaries are partly based on model definitions of the systems TGraph [55] and Graphite [27].

Definition 1 (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 $\kappa : (L \cup V \cup E) \times K \rightharpoonup A$.

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 $l : V \cup E \to \mathbb{P}(L) \setminus \{\emptyset\}$ 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* $\mathcal{G} = \{G_1, G_2, \ldots, G_n\} \subseteq \mathbb{P}(L)$ 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 $v_1 id = v_2 id \wedge v_1 \neq v_2 \implies \tau_1^{tx} \cap \tau_2^{tx} = \emptyset$. 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 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[4]. This graph consists of the vertex set $V = \{v_0, \ldots, v_4\}$ and the edge set $E = \{e_0, \ldots, e_{10}\}$. 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.

To visualize the graph's evolution, a timeline is placed below the graph in Fig. 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 $\omega_{\min}$ and maximum $\omega_{\max}$ 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 $V(G_1) \cap V(G_2) = \{\emptyset\}$ and $E(G_1) \cap E(G_2) = \{\emptyset\}$. 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.

4.2 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

[4] https://www.citibikenyc.com/system-data.

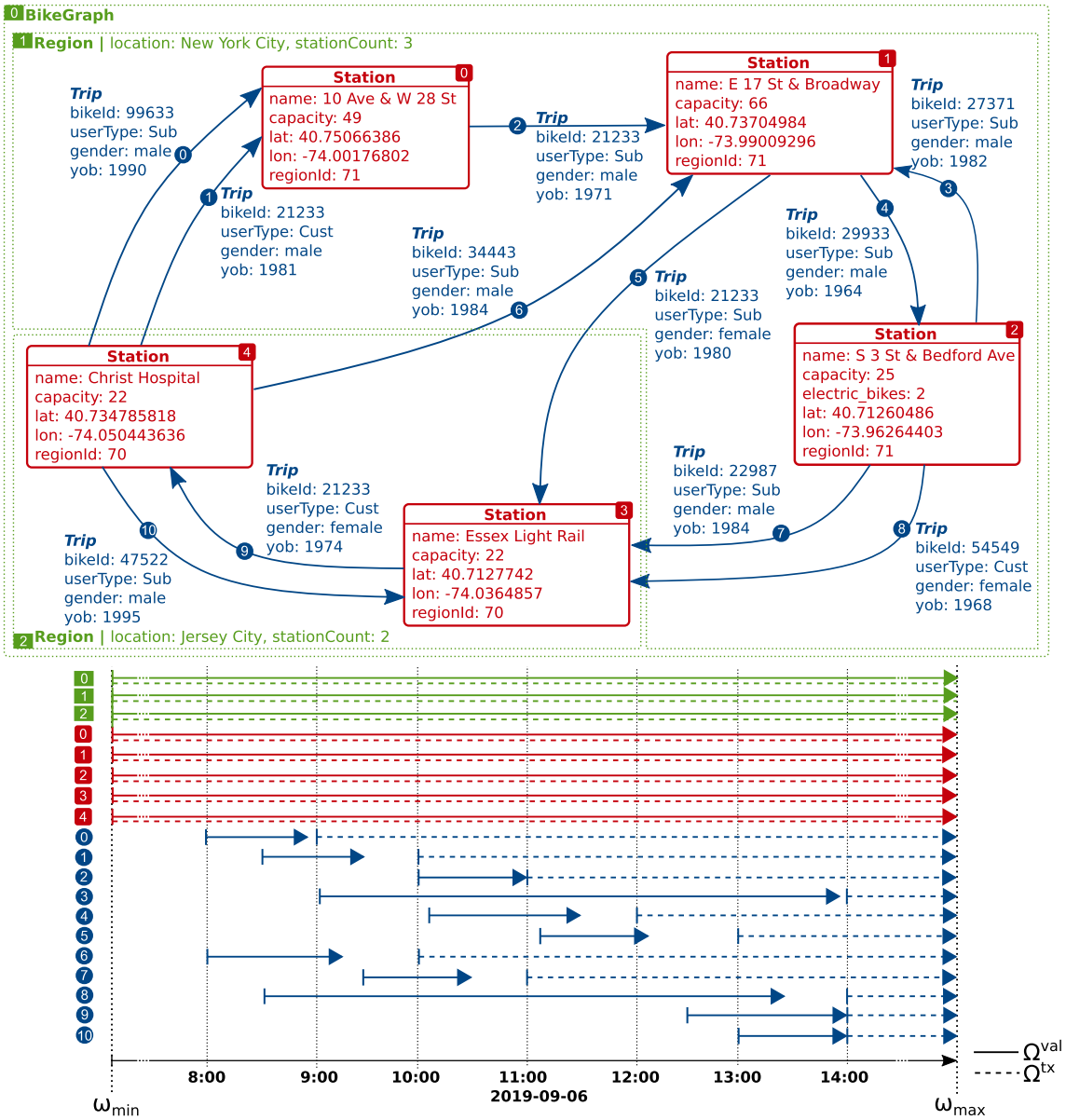

Fig. 2 Example temporal property graph of a bike rental network with two non-overlapping logical graphs

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 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 [40,41,43,65,66] and GRADOOP's GitHub–Wiki [64] for detailed explanations.

4.2.1 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

Table 2 TPGM graph operators specified with GRALA

	Operator	GrALa	
		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

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 $\varphi_v : V \rightarrow \{true, false\}$ and $\varphi_e : E \rightarrow \{true, false\}$, 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 `Trip` with a property `gender` which value is equal to `female`:[5]

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

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

```
(e => e.label == 'Trip' AND
      e.gender == 'female'))
```

Applied to the graph G_0 of Fig. 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 function exclusively, the operator is also suitable to declare vertex-induced or edge-induced subgraphs, respectively.

4.2.2 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 [47] and Allen's interval algebra [9].

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:

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

In the timeline of Fig. 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.

4.2.3 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 Fig. 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 no 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.

4.2.4 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 [40,43] and exploratory analyze 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, $V' = \{v'_1, v'_2, \ldots, v'_k\}$ where v'_i is called a *supervertex* and $\forall v \in V$, $s_v(v)$ is the supervertex of v.

Vertices are grouped together based on the values returned by *key functions*. A key function $k : V \rightarrow \mathcal{V}$ 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 $\forall u, v \in V : s_v(u) = s_v(v) \iff \bigwedge_{i=1}^{n} k_i(u) = k_i(v)$. 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 supervertex as new properties.

Similarly, $E' = \{e'_1, e'_2, \ldots, e'_l\}$ where e'_i is called a *superedge* and $s_\epsilon(u, v)$ is the superedge for $\langle u, v \rangle$. Edge groups are determined along the supervertices and a set of edge keys $\{k_1, \ldots, k_m\}$, where $k_j : E \rightarrow \mathcal{V}$ are grouping key functions analogous to the vertex keys, such that $\forall e, f \in E : s_\epsilon(s(e), t(e)) = s_\epsilon(s(f), t(f)) \iff s_v(s(e)) = s_v(s(f)) \wedge s_v(t(e)) = s_v(t(f)) \wedge \bigwedge_{j=1}^{m} k_j(e) = k_j(f)$. The same key functions mentioned above for vertices are also applicable for edges. Additionally, vertex and edge aggregate functions $\gamma_v : \mathcal{P}(\mathcal{V}) \rightarrow A$ and $\gamma_e : \mathcal{P}(\mathcal{E}) \rightarrow A$ 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 supervertex and superedge, respectively. The following example shows the application of the grouping operator using GRALA:

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

The goal of this example is to group *Station*s and *Trip*s in the graph of Fig. 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 2, 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 supervertices and superedges. In lines 3–9 and 12–15, we declare the vertex and edge aggregate functions, respectively. Both receive a superelement (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 superelement. 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 shows the resulting logical graph for this example.

4.2.5 Temporal pattern matching

A fundamental operation of graph analytics is the retrieval of subgraphs isomorphic or homomorphic[6] 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-sharing network, an analyst 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 [41], 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*[7], a query language which is based on the core concepts of Cypher[8], 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

[6] GRALA support different morphism semantics, see [41].

[7] TemporalGDL is an extension of the **G**raph **D**efinition **L**anguage (GDL) which is open source available at https://github.com/dbs-leipzig/gdl.

[8] http://www.opencypher.org/.

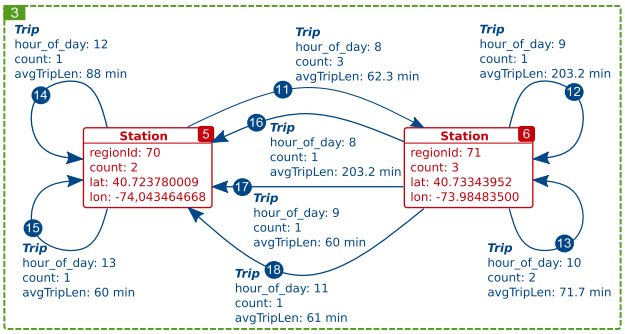

Fig. 3 Result graph of grouping example

labels and properties or expressed in the `WHERE` clause. For a more detailed description of the (non-temporal) language on which *TemporalGDL* is based, we refer to our previous publication [41]. We extended the language by various syntactic constructs that are partly inspired by Allen's conventions [9] and the SQL:2011 standard [47], 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 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 whether 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 whether the duration of the valid time interval of `a` is shorter than the one of `b` or `a.val.longerThan(Minutes(5))` to check whether 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. 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)")
```

Table 3 Overview of TemporalGDL's syntax to support temporal graph patterns

Syntax construct	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 give 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

The shown *TemporalGDL* pattern graph reflects the aforementioned bike-sharing 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

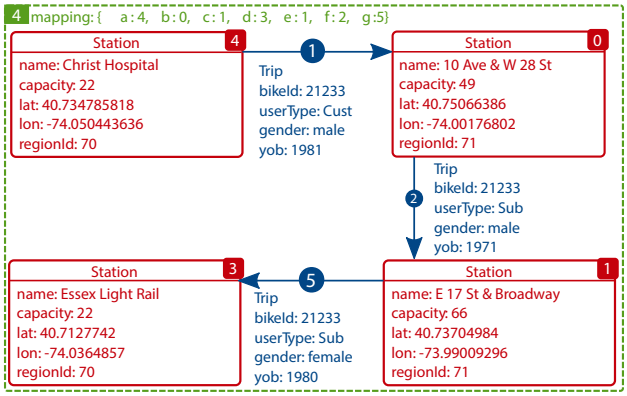

Fig. 4 Result graph of temporal pattern matching example

of Fig. 2, the operator returns a graph collection containing a single logical graph as shown in Fig. 4. Each graph in the result collection contains a new property storing the mapping between query variables and entity identifiers.

4.2.6 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., $v : V \rightarrow V$ 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 analysis [46]. 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 [46] and GRADOOP's GitHub wiki [64].

4.2.7 Additional graph operators

As shown in Table 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 [29] and the GitHub wiki [64].

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.

4.2.8 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 whether 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.

4.3 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 [48] and their evolution [30].

To support external algorithms, GRALA provides generic *call* operators (see Table 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 [3], which is the Apache Flink implementation of Google Pregel [52]. By utilizing Flink's dataset iteration, co-group and flatMap 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 Sect. 5.4. Thus, GRADOOP already provides a set of algorithms that can be seamlessly integrated into a graph analytical program (see Table 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 [61].

5 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 [84] and Apache Flink [16] is especially promising. These systems offer, in contrast to MapReduce [24], 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 Sect. 2, the most recent approaches to large-scale graph analytics are libraries on top of such distributed dataflow frameworks, e.g., GraphX [83] on Apache Spark or Gelly [3] 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, collections and are generic, which means arbitrary user-defined data can be attached to vertices and edges. In consequence, model-specific operators, i.e., such based on labels, properties or the 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 to large-scale data and graph processing at the same time. The majority of graph algorithms listed in Table 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.

5.1 Apache Flink

Apache Flink [8,16] 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 Sect. 4.2). 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 datasets 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).

5.2 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 with 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`).

5.2.1 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 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:

```
LogicalGraph graph = new
    CSVDataSource(...)
  .getLogicalGraph();

GraphCollection triangles = graph
  .snapshot(
    new Overlaps('2019-09-06',
    '2019-09-07'))
  .subgraph((e => e.yob > 1980))
  .callForGraph(new PageRank
    ('pr', 0.8, 10))
  .query("
    MATCH (p1)-->(p2)-->(p3)<--(p1)
    WHERE ((p1.pr + p2.pr + p3.pr) / 3)
      > 0.8)
  ");

new CSVDataSink(...).write(triangles);
```

We start by reading a logical graph from a specific data source. We then retrieve a snapshot with all elements that overlap the given period in the past. After that, 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 and store the resulting rank as a new vertex property `pr`. Using the match operator, 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. 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 Sect. 6, we will provide more complex examples as part of our evaluation.

5.2.2 Graph Layouts

While the two programming abstractions provide an implementation-independent access to the GRALA API, their internal Flink DataSet representations are encapsulated by specific *graph layouts*. The most common *GVE Layout*

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)
				...			

Fig. 5 GVE layout of GRADOOP. The accuracy of the timestamps has been reduced for readability reasons

(**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 not required. 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 5 shows an example instance of the GVE layout for a graph collection containing logical graphs of Fig. 2. The first DataSet L stores the data attached to logical graphs, vertex data is stored in a second DataSet V and edge data 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.

5.3 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 4 introduces a subset of transformations available in Apache Flink. Well-known transformations have been adopted from the MapReduce paradigm [24]. 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 Sect. 4: 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 Sect. 5.2.2).

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 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*

Table 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>>`)

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 flatmap 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>`

transformation between filtered edges and filtered vertices for both `sourceId` 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.[9]

Snapshot The snapshot operator [66] 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* 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.

[9] 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.

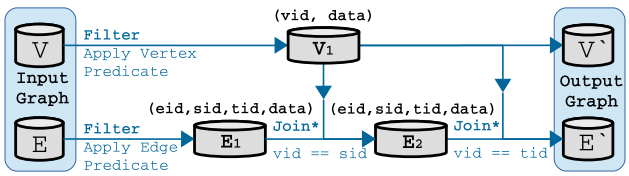

Fig. 6 Dataflow implementation of the subgraph operator using Flink DataSets and transformations

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 Sect. 4.2), which we previously introduced [66]. 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 Fig. 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 whether that element exists in both, none, only the first or only the second snapshot. It will be collected and annotated

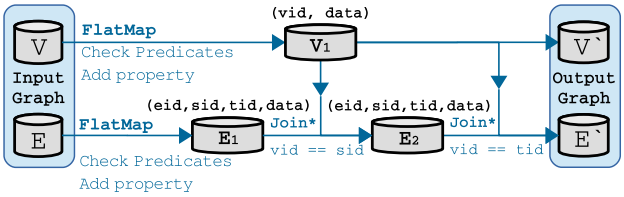

Fig. 7 Dataflow implementation of the difference operator using Flink DataSets and transformations

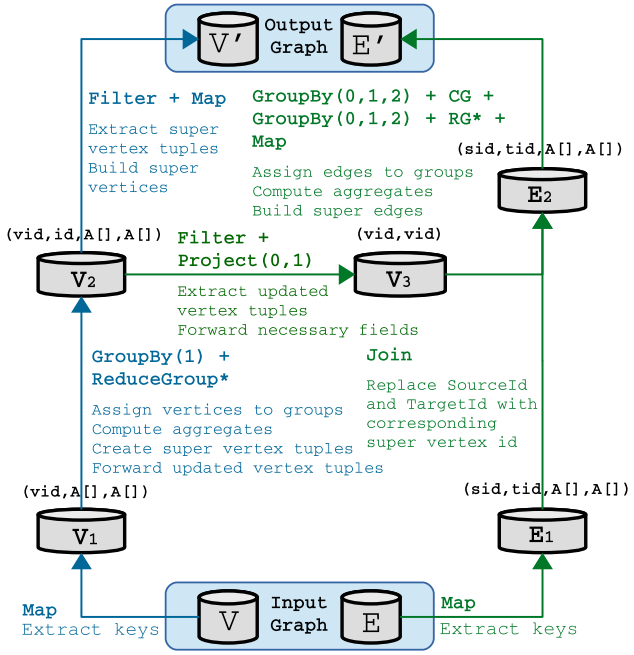

Fig. 8 Dataflow implementation of the grouping operator using Flink DataSets and transformations. Lists of property values are denoted by the type `A[]`

with a property as defined in Sect. 4.2, 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 *join*s.

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 *supervertex* and a mapping from vertices to supervertices is extracted. Edges are additionally grouped with their source and target vertex identifier. Figure 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 *supervertex tuple* for each group and (2) creating a mapping from vertices to supervertices via their identifier. The supervertex tuple has a similar structure to the vertex tuple, except that it stores the supervertex identifier, the grouping keys and calculated aggregate values for every aggregation function. In the final step for vertices, we construct supervertices 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 supervertex identifier (DataSet V_3) which can in turn be used to update the input edges in DataSet E and to group them to get superedges 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 supervertex and again to replace the target identifier. Since the resulting edges (now stored in DataSet E_2) are logically connecting supervertices, we can group them on source and target identifier as well as key function values. This step yields tuple representations of superedges, which finally mapped to the new `Edge` instances, representing the final superedges.

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 [33,41] 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 following 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 9 shows a simplified[10] dataflow program for the following temporal query:

```
MATCH (a:Station)<-[e:Trip]-(c:Station)
      (b:Station)<-[f]-(c)
WHERE e.val.asOf(Timestamp(2019-09-06)) AND
      e.val.overlaps(f.val) AND
      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 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 postprocessing 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 [41]. 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.

[10] 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.

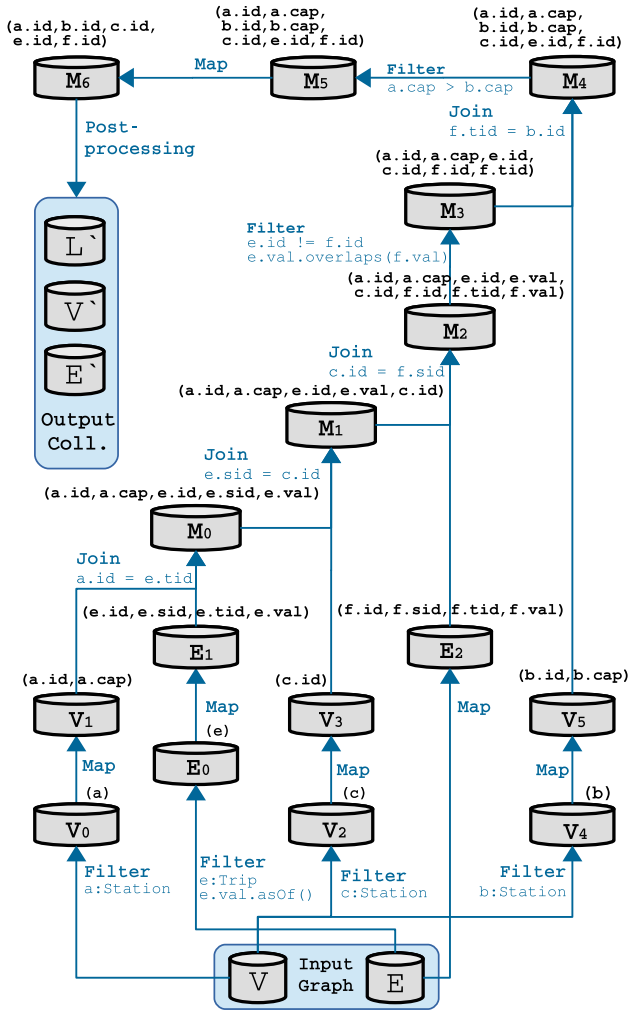

Fig. 9 Dataflow representation of a pattern matching query

5.4 Iterative graph algorithms

Gelly, the Graph API of Apache Flink, uses iteration operators to support large-scale iterative graph processing [3]. 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 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.

5.5 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 [16]. 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 Sect. 5.2.1). 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 Sect. 5.2.2). 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 postprocessing.

In addition to the file-based formats, GRADOOP supports two distributed database systems for storing logical graphs and graph collections: HBase [2] and Accumulo [1]. Both storage engines follow the BigTable approach allowing wide tables including column families and fast row lookup by primary keys [19].

Another supported format which is purely used for visualization purpose is the open graph description format DOT [28]. A detailed description of both sources and sinks can be found in GRADOOP's GitHub wiki [64].

6 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.

Table 5 Dataset characteristics

Name	SF	$\|V\|$	$\|E\|$	Disk size
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

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., p-law distribution [35]. *CitiBike* is a real-world dataset describing New York City bike rentals since 2013[11]. The schema of this dataset corresponds to the one in Fig 2 in Sect. 4. Table 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 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 logical graph. We further group the combined graph by the vertex properties `name` and `cellId` (line 17) and the

[11] https://www.citibikenyc.com/.

```
outGraph = citiBikeGraph                                    1
 .snapshot(                                                 2
   Overlaps('2017-01-01', '2019-01-01'))                    3
 .transform(                                                4
   v => v['cellId'] = getGridCellId(v))                     5
 .query(                                                    6
   "MATCH (v1:Station)-[t1:Trip]                            7
        ->(v2:Station)                                      8
          (v2)-[t2:Trip]->(v3:Station)                      9
    WHERE v1.cellId == 2883 AND                             10
          v2.id != v1.id AND                                11
          v2.id != v3.id AND                                12
          t1.val.precedes(t2.val) AND                       13
          t1.val.lengthAtLeast                              14
             (Minutes(X)) AND                               15
          t2.val.lengthAtLeast                              16
             (Minutes(X))")                                 17
 .reduce(g, h => g.combine(h))                              18
 .groupBy(                                                  19
   [label(),prop('name'),prop('cellId')],                   20
   (),                                                      21
   [label(),timestamp(val-from, MONTH)],                    22
   (superEdge, edges =>                                     23
     superEdge['count'] = edges.count(),                    24
     superEdge['avgDur']                                    25
        = edges.avgDur()))                                  26
 .subgraph(e => e['count'] > 1)                             27
```

Listing 1 CitiBike Analytical (CBA) program. (X defines the minimum duration of the bike rentals)

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 superedge and the average duration of the found rentals. Finally, we apply a *Subgraph* operator to output only superedges with a count greater than 1 (line 23). The corresponding source code is available online[12].

Figures 10, 11 and 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 10 shows the impact of the LDBC data size on the runtime of individual operators. Figures 11 and 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. 10) and cluster size (Figs. 11 and 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 12 shows the speedup for LDBC.100 grows nearly linearly until 4 parallel workers and is slightly declining then for more workers. This behavior 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 step), while the useful work per worker becomes smaller with larger configurations.

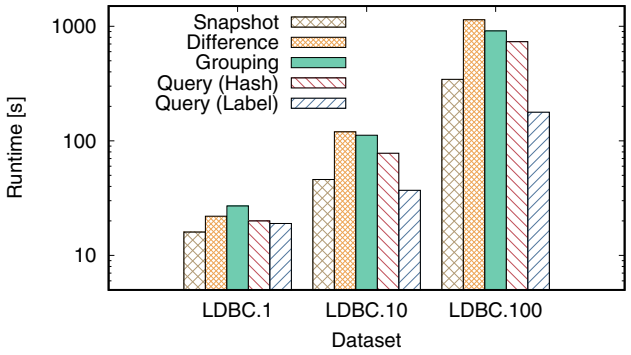

Fig. 10 Increase data volume

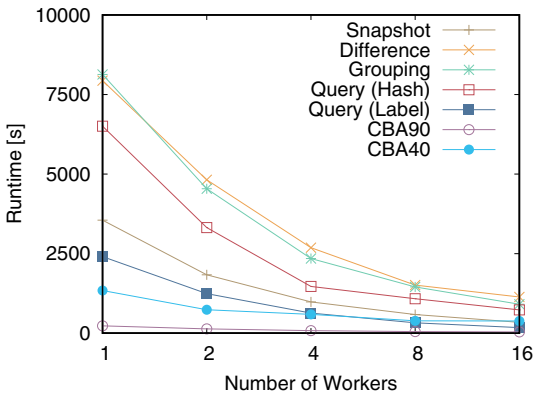

Fig. 11 Increase worker count

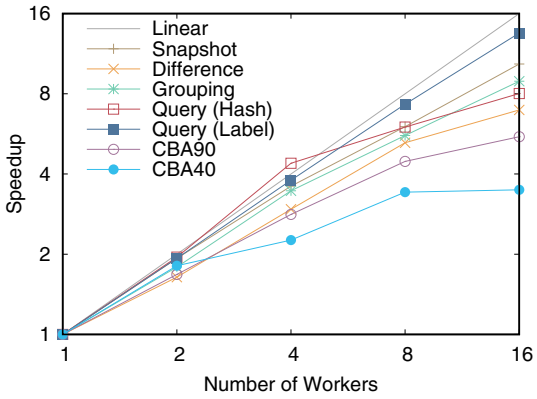

Fig. 12 Speedup over workers

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 supervertices and superedges that show the weekly evolu-

[12] https://git.io/JULPM.

tion of created entities and their respective relationships over several years. For aggregation, we count the number of new entities and relationships per week.

Figure 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 (Fig. 11) resulting in a speedup of more than 8. Figure 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 [43].

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 Sec 5.3) can lead to an unbalanced workload for skewed group sizes. We already addressed this issue in [43]; 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 predefined 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 Sect. 5.2.2). 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 >= Timestamp
            (2012-06-01) AND
      l.val_from <= Timestamp
            (2012-06-02) AND
      c.val_from >= Timestamp
            (2012-05-30) AND
      c.val_from <= Timestamp
            (2012-06-02) AND
      po.val_from >= Timestamp
            (2012-05-30) AND
      po.val_from <= Timestamp
             (2012-06-02)
```

Regarding the default hash partitioning, Fig. 10 shows that the execution of this query scales linearly with a larger data size from LDBC.10 (78 s) to LDBC.100 (733 s). Increasing the cluster size for LDBC.100 reduces the query runtime from 6857 s for one worker to 733 seconds using 16 workers (Fig. 11). The speedup behavior (Fig. 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. 10, the runtime improvements increase with growing data volume from a factor of 2 for LDBC.10 (37 instead of 78 s) to a factor of 4 for LDBC.100 (178 vs. 733 s).

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. 11) and speedup (Fig. 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.

CBA The results for the more complex analytical pipelines CBA40 and CBA90 are shown in Figs 11 and 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 10 M (CBA40) and 150 K (CBA90) matches (i.e., matching subgraphs of the query graph) representing trips with a duration of at least 40 or 90 min.

The more selective CBA90 program could be executed in 223 s on a single worker and only 42 s on 16 workers, while CBA40 takes 1336 and 352 s 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 have 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.

7 Lessons learned and ongoing work

We achieved the main goals of the GRADOOP project to develop a comprehensive open-source framework for the distributed processing and analysis of large temporal property graphs. This way we combine positive features of graph database systems and of distributed graph processing systems and extend these systems in several ways, e.g., with support for graph collections and bitemporal graphs as well as with built-in analysis capabilities including structural graph transformations and temporal graph analysis operators. Given that this project runs already over 5 years we will now reflect on some of our design decisions concerning technology selection, data model, operator concept, etc. Finally, we give an outlook to ongoing and future work.

Apache Flink One of the first and 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 its rich set of composable (Flink) transformations and support for automatic program optimization without the need for deeper system knowledge.

In recent years, however, Apache Flink has focused to become a pure stream processing engine so that the DataSet API used by GRADOOP is planned to be deprecated. We have therefore begun to evaluate alternate processing frameworks such as the DataStream and Table APIs 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. Such a major change could thus allow better support for processing, analysis and continuous queries on graph streams [15] that we plan to address in the future.

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 structure of the graph by adding new vertex or edge types or adding redundant properties. Graph collections, i.e., sets of logical graphs, have been found to be a hugely valuable data structure for modeling (possibly overlapping) logical graphs. Graph collections also facilitate the use of binary graph operations, 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.

Operator concept Another core design decision was the methodology to introduce operators that can be 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 and hidden to the analyst, thus providing a top level abstraction of the respective function. The large number of operators provided, which contain both simple analyzes and graph algorithms, can therefore be used as a tool kit 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. 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. As a consequence, performance issues with specific operators and its dataset transformations are hard to identify.

Temporal extensions The evolution of entities and relationships is a natural characteristic of many graph datasets that represents a real-world scenario [73]. A static graph model like the PGM or the initial EPGM of GRADOOP is in most cases unsuitable for performing analyzes 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) was covering many requirements of frequently used temporal analysis, for example, the retrieval of a snapshot, without building a completely new model and prototypical implement a new framework. 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 analyzing their evolution with the grouping operator and its temporal aggregations. In future work, we plan to

develop alternatives to the GVE data layout without separating vertices and edges so that TPGM integrity conditions can be checked more efficiently. We will also investigate the separation of the newest graph state from its history for increased performance.

Scalability The evaluations so far showed that GRADOOP generally scales very well with increasing dataset sizes. On the other hand, the speedup when increasing the number of machines on a fixed dataset reaches its limit relatively fast. (For the considered workloads and datasets, the speedup was mostly lower than 10 with the default hash-based data partitioning.) A main reason for this behavior can be seen in the strong dependency on the underlying Flink system and its optimizer and scheduler that are not tailored to graph data processing. For example, data distribution is by default based on a hash partitioning, in particular for intermediate results in an analytical pipeline that prevents the utilization of data locality for our graph operators but can result in large communication overhead even for traversing edges. For that reason, 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 as well as for analysis pipelines.

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 [67] and applications and serves as basis for other research projects, e.g., on knowledge graphs [72].

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 [56]. The implementation of GRADOOP's pattern matching operator [41] as a proof of concept for the distributed execution of Cypher(-like) queries, directly influenced the development of Neo4j's Morpheus[13] 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[14]. 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 in its GitHub wiki [64] 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, whereby it can be difficult to find out which special 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 a detailed documentation in the GitHub wiki [64] to assist for finding the right combination and configuration.

Ongoing work 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[15] [7], to process temporal graphs. The continuous analysis of graph streams, where events indicate vertex and edge additions, deletions and modifications, is an emerging research area [17] and will be considered in our future research. Besides, we will integrate machine learning (ML) methods on temporal and streaming graphs, e.g., to identify diminishing and recurring patterns or to predict further evolution steps. Another area that we will work more on is to extend GRADOOP to realize temporal knowledge graphs integrating data from different sources and to continuously evolve the integrated data. A further extension is the addition of layouting algorithms for the visualization of logical graphs and graph collections to offload the expensive calculation from frontend application. Finally, we will investigate more in overall performance optimization of GRADOOP and add temporal graph algorithms as further operators to the framework.

8 Conclusion

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

[13] https://github.com/opencypher/morpheus.

[14] https://mvnrepository.com/artifact/org.gradoop.

[15] https://statefun.io.

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 were able to speed up the performance of our pattern matching operator by a large margin. We finally reflect on several lessons learned during our 6-year experience working on that project. 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, and (iii) the integration of analysis using graph ML.

Acknowledgements This work is partially funded by the German Federal Ministry of Education and Research under grant BMBF 01IS18026B in project ScaDS.AI Dresden/Leipzig.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Apache Accumulo, July 2020. https://accumulo.apache.org/
2. Apache HBase, July 2020. http://hbase.apache.org/
3. Gelly: Flink Graph API, July 2020. https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/gelly/
4. JanusGraph, July 2020. https://janusgraph.org/
5. OrientDB Community, July 2020. http://www.orientechnologies.com/orientdb/
6. Aghasadeghi, A., Moffitt, VZ, Schelter, S., Stoyanovich, J.: Zooming out on an evolving graph. In: Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, vol. 23, pp. 25–36 (2020)
7. Akhter, A., Fragkoulis, M., Katsifodimos, A.: Stateful functions as a service in action. Proc. VLDB Endow. **12**(12), 1890–1893 (2019)
8. Alexandrov, A., et al.: The stratosphere platform for big data analytics. The VLDB J. **23**(6), 939–964 (2014)
9. Allen, J.F.: Maintaining knowledge about temporal intervals. Commun. ACM **26**(11), 832–843 (1983)
10. Angles, R.: A Comparison of Current Graph Database Models. In: Proc, ICDEW (2012)
11. Angles, R.: The property graph database model. In: AMW, volume 2100 of *CEUR Workshop Proceedings*. CEUR-WS.org, (2018)
12. Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J., Vrgoč, D.: Foundations of modern query languages for graph databases. ACM Comput. Surv. (CSUR) **50**(5), 1–40 (2017)
13. Batarfi, O., et al.: Large scale graph processing systems: survey and an experimental evaluation. Cluster Comput. **18**(3), 1189–1213 (2015)
14. Berhold, M.R. et al.: KNIME - the Konstanz Information Miner: Version 2.0 and Beyond. SIGKDD Explor. Newsl., (2009)
15. Besta, M., Fischer, M., Kalavri, V., Kapralov, M., Hoefler, T.: Practice of streaming and dynamic graphs: Concepts, models, systems, and parallelism. arXiv preprint arXiv:1912.12740 (2019)
16. Carbone, P., et al.: Apache Flink: Stream and Batch Processing in a Single Engine. IEEE Data Eng. Bull. **38**(4), 28–38 (2015)
17. Carbone, P., Fragkoulis, M., Kalavri, V., Katsifodimos, A.: Beyond analytics: The evolution of stream processing systems. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp. 2651–2658 (2020)
18. Casteigts, A., Flocchini, P., Quattrociocchi, W., Santoro, N.: Time-varying graphs and dynamic networks. Int. J. Parallel Emergent Distrib. Syst. **27**(5), 387–408 (2012)
19. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A distributed storage system for structured data. ACM Trans. Comput. Syst. **26**(2), 4:1–4:26 (2008)
20. Chao, S.-Y: Graph theory and analysis of biological data in computational biology, Chapter 7. In: Advanced technologies. InTech (2009)
21. Cheng, R., Hong, J., Kyrola, A., Miao, Y., Weng, X., Wu, M., Yang, F., Zhou, L., Zhao, F., Chen, E.: Kineograph: taking the pulse of a fast-changing and connected world. In: Proceedings of the 7th ACM European Conference on Computer Systems, pp. 85–98, (2012)
22. Corp, O.: Oracle's Graph Database. https://www.oracle.com/de/database/graph/
23. Curtiss, M., et al.: Unicorn: A system for searching the social graph. PVLDB **6**(11), 1150–1161 (2013)
24. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Commun. ACM **51**(1), 107–113 (2008)
25. Erb, B., Meißner, D., Pietron, J., Kargl, F.: 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, pp. 78–87, (2017)
26. Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., Taylor, A.: Cypher: An evolving query language for property graphs. In: Proceedings of ACM SIGMOD, pp. 1433–1445, (2018)
27. Gandhi, S., Simmhan, Y.: An interval-centric model for distributed computing over temporal graphs. In: 2020 IEEE 36th International Conference on Data Engineering (ICDE), pp. 1129–1140. IEEE, (2020)
28. Gansner, E., Koutsofios, E., North, S.: Drawing graphs with dot. http://www.graphviz.org/pdf/dotguide.pdf
29. Gomez, K., Täschner, M., Rostami, M.A., Rost, C., Rahm, E.: Graph sampling with distributed in-memory dataflow systems (2019) arXiv preprint arXiv:1910.04493
30. Gong, M.-G., Zhang, L.-J., Ma, J.-J., Jiao, L.-C.: Community detection in dynamic social networks based on multiobjective immune algorithm. J. Comput. Sci. Technol. **27**(3), 455–467 (2012)
31. Guo, Y., Biczak, M., Varbanescu, A.L., Iosup, A., Martella, C., Willke, T.L.: How well do graph-processing platforms perform? an empirical performance evaluation and analysis. In: Proceedings of IPDPS (2014)
32. Holme, P., Saramäki, J.: Temporal networks. Phys. Rep. **519**(3), 97–125 (2012). Temporal Networks
33. Hölsch, J., Grossniklaus, M.: An Algebra and Equivalences to Transform Graph Patterns in Neo4j. In: Proceedings of EDBT Workshops (2016)

34. Huang, H., Song, J., Lin, X., Ma, S., Huai, J.: Tgraph: A temporal graph data management system. In: Proceedings of the 25th ACM International on Conference on Information and Knowledge Management, pp. 2469–2472 (2016)
35. Iosup, A., et al.: Ldbc graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. Proc. VLDB Endow. **9**(13), 1317–1328 (2016)
36. Iyer, A., Pu, Q., Patel, K., Gonzalez, J., Stoica, I.: Tegra: Efficient ad-hoc analytics on time-evolving graphs. Technical Report, UCBerkeley RISELab (2019)
37. Jensen, C.S., Snodgrass, R.T.: Temporal data management. IEEE Trans. Knowl. Data Eng. **11**(1), 36–44 (1999)
38. Johnston, T.: Bitemporal Data: Theory and Practice. Newnes, New South Wales (2014)
39. Junghanns, M., et al.: GRADOOP: scalable graph data management and analytics with hadoop. CoRR, (2015)
40. Junghanns, M., et al.: Analyzing Extended Property Graphs with Apache Flink. In: Proceedings of SIGMOD NDA Workshop (2016)
41. Junghanns, M., Kießling, M., Averbuch, A., Petermann, A., Rahm, E.: Cypher-based graph pattern matching in gradoop. In: Proceedings of GRADES (2017)
42. Junghanns, M., Petermann, A., Neumann, M., Rahm, E.: Management and Analysis of Big Graph Data: Current Systems and Open Challenges. Springer, In: Handbook of Big Data Technologies (2017)
43. Junghanns, M., Petermann, A., Rahm, E.: Distributed Grouping of Property Graphs with GRADOOP. In Proc: BTW (2017)
44. Klyne, G., Carroll, J.J., McBride, B.: Resource description framework (RDF): Concepts and abstract syntax. http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/
45. Kostakos, V.: Temporal graphs. Physica A **388**(6), 1007–1023 (2009)
46. Kricke, M., Peukert, E., Rahm, E.: Graph Data Transformations in Gradoop. In: BTW, volume P-289 of *LNI*, pp. 193–202, (2019)
47. Kulkarni, K., Michels, J.-E.: Temporal features in sql: 2011. ACM Sigmod Record **41**(3), 34–43 (2012)
48. Lancichinetti, A., Fortunato, S.: Community detection algorithms: a comparative analysis. Phys. Rev. E **80**(5), 056117 (2009)
49. Lazarevic, L.: Keeping track of graph changes using temporal versioning, (December 2019). https://medium.com/neo4j/keeping-track-of-graph-changes-using-temporal-versioning-3b0f854536fa
50. Lightenberg, W., Pei, Y., Fletcher, G., Pechenizkiy, M.: Tink: A temporal graph analytics library for apache flink. Companion Proc. The Web Conf. **2018**, 71–72 (2018)
51. Ling, Z.J., et al.: GEMINI: An integrative healthcare analytics system. PVLDB **7**(13), 1766–1771 (2014)
52. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: Proc. SIGMOD (2010)
53. McColl, R., et al.: A performance evaluation of open source graph databases. In: Proc. PPAA (2014)
54. Miao, Y., Han, W., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, E., Chen, W.: Immortalgraph: A system for storage and analysis of temporal graphs. ACM Trans. Storage (TOS) **11**(3), 1–34 (2015)
55. Moffitt, V.Z., Stoyanovich, J.: Temporal graph algebra. In: Proceedings of The 16th International Symposium on Database Programming Languages, pp. 1–12, (2017)
56. Neo4j Inc. APOC Documentation - Graph Grouping. https://neo4j.com/labs/apoc/4.1/virtual/graph-grouping/
57. Neo4j Inc. Neo4j. http://www.neo4j.com/
58. Neo4j Inc. Cypher Query Language, (July 2020). https://neo4j.com/developer/cypher/
59. Obraczka, D., Saeedi, A., Rahm, E.: Knowledge graph completion with FAMER. In: Proc. KDD DI2KG workshop, (2019)
60. Petermann, A., et al.: BIIIG: Enabling Business Intelligence with Integrated Instance Graphs. In: Proc. ICDEW (2014)
61. Petermann, A., et al.: Dimspan - transactional frequent subgraph mining with distributed in-memory dataflow systems. Proc. BDCAT (2017)
62. Rodriguez, M.A.: The gremlin graph traversal machine and language (invited talk). In: Proceedings of the 15th Symposium on Database Programming Languages, pp. 1–10, (2015)
63. Rodriguez, M.A., Neubauer, P.: Constructions from Dots and Lines. arXiv:1006.2361v1, (2010)
64. Rost, C., Fritzsche, P., Gomez, K., Adameit, T., Schons, L.: GitHub Wiki of Gradoop, (July 2020). https://github.com/dbs-leipzig/gradoop/wiki
65. Rost, C., Thor, A., Fritzsche, P., Gómez, K., Rahm, E.: Evolution Analysis of Large Graphs with Gradoop. In: PKDD/ECML Workshops (1), volume 1167 of Communications in Computer and Information Science, pages 402–408. Springer, (2019)
66. Rost, C., Thor, A., Rahm, E.: Analyzing temporal graphs with Gradoop. Datenbank-Spektrum **19**(3), 199–208 (2019)
67. Rostami, M.A., Kricke, M., Peukert, E., Kühne, S., Wilke, M., Dienst, S., Rahm, E.: BIGGR: Bringing Gradoop to applications. Datenbank-Spektrum **19**, 51–60 (2019)
68. Rostami, M.A., Peukert, E., Wilke, M., Rahm, E.: Big graph analysis by visually created workflows. In: Grust, T., Naumann, F., Böhm, A., Lehner, W., Härder, T., Rahm, E., Heuer, A., Klettke, M., Meyer, H. editors, BTW 2019, pp. 559–563, (2019)
69. Rudolf, M., Paradies, M., Bornhövd, C., Lehner, W.: 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-loated with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013, p. 16 (2013)
70. Saeedi, A., Nentwig, M., Peukert, E., Rahm, E.: Scalable matching and clustering of entities with FAMER. Complex Syst. Inf. Modeling Q. **16**, 61–83 (2018)
71. Saeedi, A., Peukert, E., Rahm, E.: Using link features for entity clustering in knowledge graphs. In: European Semantic Web Conference, pp. 576–592. Springer, (2018)
72. Saeedi, A., Peukert, E., Rahm, E.: Incremental multi-source entity resolution for knowledge graph completion. In: European Semantic Web Conference, pp. 393–408. Springer, (2020)
73. Sahu, S., Mhedhbi, A., Salihoglu, S., Lin, J., Özsu, M.T.: The ubiquity of large graphs and surprising challenges of graph processing: extended survey. VLDB J. **29**(2–3), 595–618 (2020)
74. Semertzidis, K., Pitoura, E.: Time traveling in graphs using a graph database. In: EDBT/ICDT Workshops (2016)
75. Shao, B., Wang, H., Xiao, Y.: Managing and mining large graphs: Systems and implementations. In: Proc. SIGMOD (2012)
76. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop Distributed File System. In: Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), MSST '10, (2010)
77. Steer, B., Cuadrado, F., Clegg, R.: Raphtory: Streaming analysis of distributed temporal graphs. Future Gener. Computer Syst. **102**, 453–464 (2020)
78. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)
79. van Rest, O., Hong, S., Kim, J., Meng, X., Chafi, H.: Pgql: a property graph query language. In: Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, p. 7. ACM, (2016)

80. Wang, Y., Yuan, Y., Ma, Y., Wang, G.: Time-dependent graphs: Definitions, applications, and algorithms. Data Sci. Eng. **4**(4), 352–366 (2019)
81. Wang, Z., Fan, Q., Wang, H., Tan, K., Agrawal, D., El Abbadi, A.: Pagrol: Parallel graph OLAP over large-scale attributed graphs. In: Proc. ICDE (2014)
82. Wu, H., Cheng, J., Huang, S., Ke, Y., Lu, Y., Xu, Y.: Path problems in temporal graphs. Proc. VLDB Endow. **7**(9), 721–732 (2014)
83. Xin, R.S., et al.: GraphX: A resilient distributed graph system on spark. In: Proc. GRADES (2013)
84. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, pages 2. USENIX Association, (2012)