Datenbank Spektrum (2019) 19:199–208
https://doi.org/10.1007/s13222-019-00325-8

SCHWERPUNKTBEITRAG

Analyzing Temporal Graphs with Gradoop

Christopher Rost[1] · Andreas Thor[2] · Erhard Rahm[1]

Received: 18 June 2019 / Accepted: 6 September 2019 / Published online: 16 September 2019

Abstract
The temporal analysis of evolving graphs is an important requirement in many domains but hardly supported in current graph database and graph processing systems. We therefore have started with extending the distributed graph analysis framework *Gradoop* for temporal graph analysis by adding time properties to vertices, edges and graphs and using them within graph operators. We outline these extensions and illustrate their use within analysis workflows. We further describe the implementation of the *snapshot* and *diff* operators and evaluated them.

Keywords Temporal Property Graph · Temporal Graph Data Model · Evolving Graph Analysis

1 Introduction

The flexible analysis of graph data has gained significant interest in the last decade and is supported by graph database systems (e. g., Neo4j) and a growing number of distributed graph processing systems [6]. As graphs typically evolve continuously, graph processing systems mostly focus on the analysis of static graphs representing the state (or snapshot) of a graph at a specific point in time. Changes like the addition of vertices and edges can occur comparatively slowly (e. g., in bibliographic networks) or at high frequency (e. g., as a stream of posts in a social network). An important requirement in many domains is to analyze the temporal dimension of graphs, e. g., to analyze the evolution of certain relationships like the citation patterns of publications or the development of co-authorships in bibliographic networks. For streaming-like changes there are specific analysis requirements, in particular to support fast, real-time reaction to certain changes such as the spread of hate messages in social networks.

This paper is an extended version of *Temporal Graph Analysis using Gradoop* [14]. We report on work in progress on temporal graph analysis using GRADOOP [5, 8], a distributed, open source framework for graph analytics build on top of Apache Flink[1]. It supports extended property graphs as well as many declarative operators, e. g., for pattern matching and structural grouping, that can be used to realize complex workflows for graph analysis. Inspired by the temporal extensions in SQL:2011 [10] we extend the GRADOOP graph data model by time properties for valid and transactional time. We also show how these temporal properties can be used within the operators for temporal graph analysis. Furthermore, we describe the implementation of two temporal operators, *snapshot* and *diff*, and evaluate their runtime efficiency and scalability for different datasets.

After a discussion of related work (Section 2) we introduce the temporal extensions of GRADOOP's property graph model (Section 3) and its operators (Section 4). We then show the use of the operators in building blocks for common tasks in temporal graph analytics (Section 5). In Section 6, we describe the implementation of the *snapshot* and *diff* operators and in Section 7 we present an evaluation of these operators.

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

Andreas Thor
thor@hft-leipzig.de

Erhard Rahm
rahm@informatik.uni-leipzig.de

[1] University of Leipzig, Leipzig, Germany

[2] Leipzig University of Telecommunications, Leipzig, Germany

[1] https://flink.apache.org/

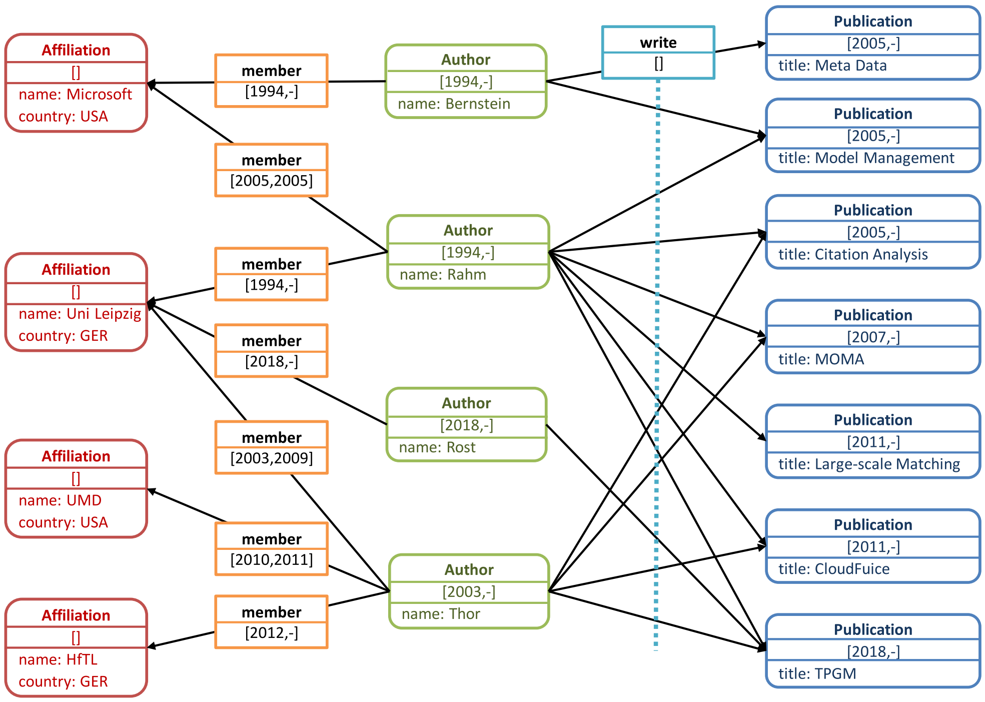

Fig. 1 TPGM example of a bibliographic graph. Vertices and edges are specified by label, valid-time (with format [*val-from,val-to*]) and properties

2 Related Work

Date et al. [2] define three classes of time aspects for temporal relational databases that can be applied one-to-one to temporal graphs: *transaction time*, *valid time*, and their combination *bitemporal data*. *Transaction time* is defined as the time interval in which a fact is considered true in the database (graph). In most cases, the transaction time is maintained by the processing system itself and can be used for versioning so that graph states can be reconstructed for any point in the past. The *valid time* is defined as a time interval in which a fact is valid as defined by the context of the application [2]. Valid time intervals can also be expressed by *time-stamps* if the duration can be neglected. Such graphs are also known as *transient* [9] or *contact sequences* [4] since their time-stamps reflect a chronological order of interactions (e.g., edge additions). Fig. 1 shows an example of a temporal graph with valid times: The temporal affiliations of authors to institutions is represented as time intervals whereas the valid time of publications is the time-stamp when the publication was published.

There are only few graph processing systems that natively support the storage, analysis and querying of temporal graphs. *Immortalgraph* [12] (earlier known as *Chronos*) provides a storage and execution engine designed for temporal graphs. It includes locality optimizations and an in-memory iterative graph computation based on series of graph snapshots. Snapshots are divided into groups to provide temporal graph mining approaches. *Kineograph* [1] is a distributed platform for incoming stream data to construct a continuously changing graph. It is also based on in-memory graph snapshots which are evaluated by conventional mining approaches of static graphs (e.g., community detection). The snapshot approach is used to distribute the graph on different systems. *GraphStream* [13] is an open-source Java library focusing on the dynamics aspects of a graph. It provides a flexible way to build user-defined analyses upon a dynamic graph structure based on a stream of graph events. None of these systems is based on a property graph model to hold detailed contextual information of vertices and edges besides the structural information. One approach of a temporal property graph model is presented by Steer et al. [15]. It is implemented in a distributed graph management system that manages the graph history in-memory. It allows updates only via event streams and offers no analysis and mining functionalities. Then et al. [16] developed an automatic algorithm transformation to avoid multiple executions of graph analytic algorithms on snapshots of a temporal graph to reduce their runtimes.

A fairly new temporal graph analytics library is *Tink* [11] that focuses on several temporal path problems and offers the calculation of measures like temporal betweenness and closeness. Similar to GRADOOP, Tink is build on Apache Flink and employs the Property Graph Model. Temporal information is represented by time intervals at the edges. In contrast to Tink, GRADOOP supports not only graphs but also logical graphs and graph collections. Furthermore, our extension of the property graph model allows the definition of both time-stamps and intervals on both vertices and edges.

Table 1 Overview of TPGM unary graph operators, their signature and output. Operators marked with * already exist in EPGM and were extended by temporal support

Operator	Signature	Output
Transformation*	`Graph.transform(graphFunction, vertexFunction, edgeFunction)`	`Graph`
Subgraph*	`Graph.subgraph(vertexPredicateFunction, edgePredicateFunction)`	`Graph`
Aggregation*	`Graph.aggregate(aggregateFunction [, ...])`	`Graph`
Snapshot	`Graph.snapshot(temporalPredicateFunction)`	`Graph`
Difference	`Graph.diff(temporalPredicateFunction, temporalPredicateFunction)`	`Graph`
Grouping*	`Graph.groupBy(vertexGroupingKeys, vertexAggregateFunction, edgeGroupingKeys, edgeAggregateFunction)`	`Graph, Collection`
Pattern Matching*	`Graph.query(patternGraph [,constructionPattern])`	`Collection`

Table 2 Predefined TPGM predicate functions that can be used by the operators. Variables x and y are timestamps, whereas c is a graph element. The predicates differ according to the definition of a time-stamp or a time-interval

Function	Predicate	
	Time-Stamp (only *from* defined)	Time-Interval (*from* and *to* defined)
asOf(x)	$from \leq x$	$from \leq x \wedge to \geq x$
fromTo(x, y)	–	$from < y \wedge to > x$
between(x, y)	–	$from \leq y \wedge to > x$
createdIn(x, y)	$from \geq x \wedge from \leq y$	$from \geq x \wedge from \leq y$
deletedIn(x, y)	–	$to \geq x \wedge to \leq y$
precedes(c)	$from \leq c.from$	$to \leq c.from$
succeeds(c)	$from \geq c.from$	$from \geq c.to$
overlaps(c)	–	$max(from, c.from) < min(to, c.to)$

3 Temporal Property Graph Model

The *Temporal Property Graph Model* (TPGM) is a simple but powerful extension of the *Extended Property Graph Model* (EPGM) [8] to support an algebra of combinable analytical operators on directed graphs that evolve over time in GRADOOP. In the EPGM, a single property graph is referred to as *logical graph*, which in turn can be part of a *graph collection*. Vertices and edges refer to one or more logical graphs and are accordingly part of them. Logical graphs, vertices, and edges consist of a unique identifier, a type label (e. g., *User* or *worksAt*), and a (possibly empty) set of properties represented as key-value pairs.

TPGM extends EPGM by adding four additional time attributes as obligatory to the schema of vertices, edges, and logical graphs: *tx-from*, *tx-to* and *val-from* and *val-to*. The first two represent the transaction time (prefix *tx*), the last two define the valid time (prefix *val*) by holding the beginning and end of the elements validity. This approach offers a flexible representation of temporal graphs with bitemporal time semantics where the valid time can be empty, a time-stamp or a time interval by setting either none, only the *val-from* or both *val-from* and *val-to* attributes. Since time attributes can be empty, also edge-centric scenarios where a graph only has time information at its edges can be modeled. Empty time attributes are interpreted as *NULL* values (e. g., in predicate functions) analogous to SQL. Fig. 1 shows a temporal graph from the bibliographic domain modeled in TPGM. The graph represents the relationship between authors, affiliations and publications. There are vertices and edges without a temporal specification (e. g., *Affiliation*), with a time-stamp (e. g., *Publication*) and a time interval (*member*) as valid-time. TPGM does not specify the data type of the time attributes and leaves it up to the implementation (e. g., Unix time-stamp or formatted date/time string). By holding a whole graph with both rollback and historical information, this model offers a flexible retrieval of arbitrary graph snapshots and dissociates itself from widespread snapshot approaches.

Valid times are typically embedded within the context of the application before they enter GRADOOP. The respective timestamps can be extracted while loading the elements as graph or collection into the system. The transaction times are maintained by the GRADOOP system automatically. Vertices and edges can be added to (or deleted from) a logical graph as well as a whole logical graph can be added to (or deleted from) a graph collection. For newly added graph elements the value of *tx-from* is set to the current system time (i. e., import time) and *tx-to* to a maximum value. If a graph element is deleted, the value of *tx-to* is set to the current system time. The deletion of a vertex automatically triggers the deletion of all corresponding edges, since they are not valid without the vertex. In addition, the model supports updates of an element's label, valid times or properties. The history of updates is also maintained by the transaction time attributes. An update is realized as a logical deletion of the updated element (i. e., the value of *tx-to* is set to the update transaction time) and a logical addition of the element including the updates (i. e., the value of *tx-from* is set to the update transaction time and *tx-to* to a maximum value). As a constraint, only graph elements that are not logically deleted (i. e., current facts) can be updated.

Another advantage of TPGM is its backward compatibility to the original EPGM since every EPGM operator can

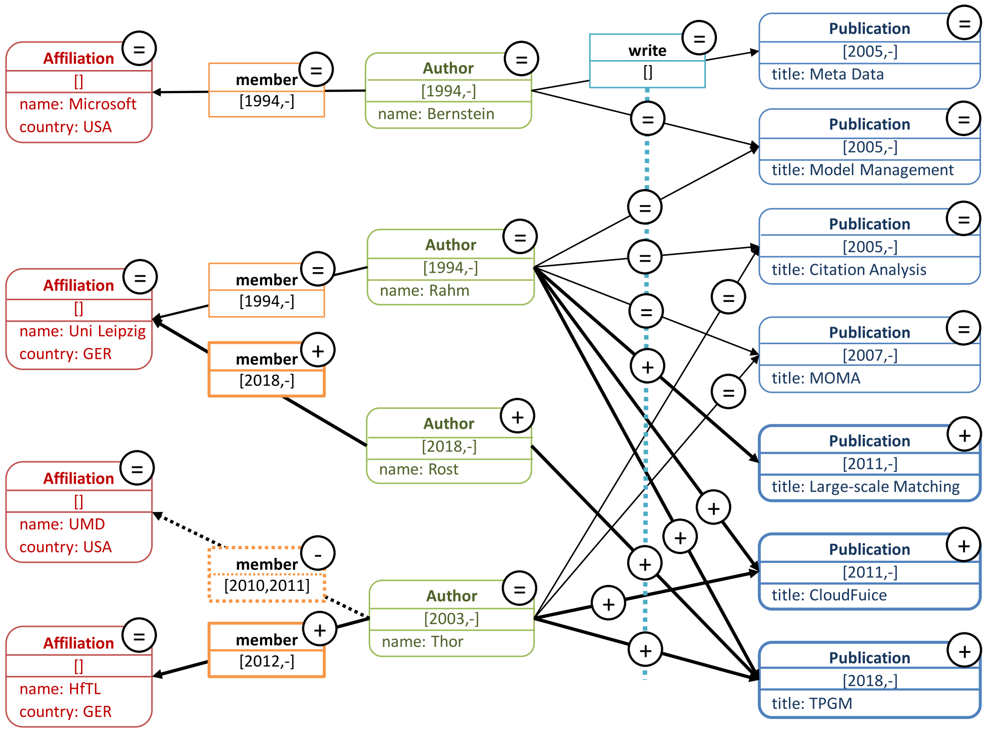

Fig. 2 The resulting annotated graph of the *Difference* operator. Vertices and edges are specified by label, valid-time (with format [*val-from,val-to*]) and properties

be applied to a TPGM graph by disregarding the temporal information of the graph elements. However, we will define appropriate operator extension for TPGM that allows the definition of temporal graph analytical workflows in the next section. Although TPGM offers bitemporal support, we limit ourselves exclusively to the valid-times for simplicity in the following sections.

4 Operators

The EPGM allows for combining multiple operators to graph analytical workflows using the domain specific language GrALa (**Gr**aph **A**nalytical **La**nguage). It already provides operator implementations for graph pattern matching, subgraph extraction, graph transformation, set operations on multiple graphs as well as property-based aggregation and selection. Some operators can be applied on logical graphs and others on graph collections [8]. For simplicity we use the terms *graph* and *logical graph*, *collection* and *graph collection* interchangeably.

The operators are implemented by a composition of Apache Flink transformations. As an example, the EPGM *subgraph* operator is realized by using the *filter* transformation of Flink. In this work we will focus on the TPGM graph operators that are listed in Table 1. They support the access and modification of the available temporal information in different ways. The following sections provide short definitions of these temporal operators with some examples. Pre-defined predicate functions that can be used by these operators are defined in Table 2. Furthermore, helper functions that return parts of a date or time information (e.g., year or day of week) are available. The implementation of the operators and their integration into GRADOOP is currently work in progress. Since we focus on valid-times in this section, each notation of *from* and *to* refers to the attributes *val-from* and *val-to* defined in TPGM.

Transformation. The *transform* operator defines a structure-preserving modification of graph, vertex and edge data. User-defined transformation functions can be applied to an input graph G, which results in an output graph G' [8]. Within TPGM it is possible to (1) modify the temporal attributes, (2) define the time attributes from information stored in properties or (3) create properties resulting from the temporal information of the time attributes. For example, if the temporal attributes are not yet set or calculated during a workflow, this operator offers the possibility to define the valid times *from* and *to* at runtime.

Subgraph. The *subgraph* operator is used to extract a subgraph from a graph by applying predefined or user-defined predicate functions [8]. Often, such a function is used to filter vertices and edges by label or the existence or value of a property (e.g., vertices with label *User* and property *age* greater than *30*). Within TPGM, the temporal information of graph elements can be used inside the predicate functions. The operator is also suitable to declare vertex-induced or edge-induced subgraphs by providing either a vertex *or* edge predicate function. Considering the graph in

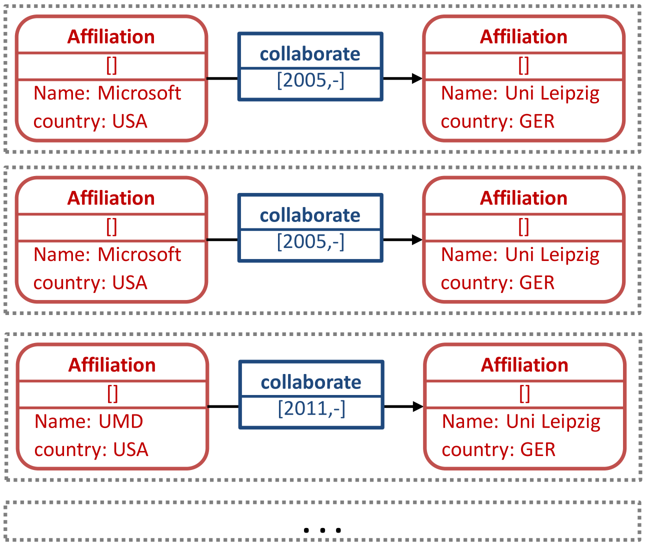

Fig. 3 Part of a graph collection containing matching subgraphs of a temporal pattern, each as a logical graph

Fig. 1, a subgraph with all author-affiliation memberships that last longer that 3 years can be extracted with the edge-induced operator call:

```
graph.subgraph(
   v -> true,
   e -> e.label = 'member'
        AND YEAR(e.to)-YEAR(e.from) > 3)
```

Aggregation. The *aggregation* operator performs one or more global aggregations on the graph. An aggregation is specified by a pre- or user-defined aggregate function. Each can be configured to be applied exclusively on vertices, edges or both. Several aggregation functions are available in the current TPGM implementation (e.g., *MinFrom* or *MaxTo*). They are realized internally by Flink's *GroupCombine* transformation. As an optimization step, we apply all given aggregate functions in *one* groupCombine step per graph element type (i.e., vertex and edge). The result of each aggregation is stored as a property on the logical graph instance. In the example of Fig. 1, the *AvgDuration* aggregate function can be used to determine the average duration of all membership edges.

Snapshot. The *snapshot* operator allows one 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 by providing a temporal predicate function. Besides the operator itself, several predefined predicate functions (see Table 2) are available. They are adopted from SQL:2011 [10] that supports temporal databases. In the example of Fig. 1, `graph.snapshot(asOf(2010))` would remove the author named *Rost* and the last three publications together with their edges. Furthermore, three member edges (*Rahm-Microsoft*, *Thor-Uni Leipzig*, and *Thor-HfTL*) are removed, too.

Difference. The evolution of graphs over time can be represented by the difference of two graph snapshots, i.e., by a difference graph that is the union of both snapshots where each graph element is annotated as an added, deleted, or persistent element. To this end, GRADOOP's structural *diff* operator consumes two graph snapshots defined by temporal predicate functions and calculates the difference graph. For example, the usage of `graph.diff(asOf(2010),asOf(2018))` at the graph in Fig. 1 would result in the annotated graph at Fig. 2. The symbols +, `-` and = represent added, removed and persisting elements, respectively.

Grouping. A structural grouping of vertices and edges is an important task in temporal graph analytics. Since temporal graphs can become very large, a condensation can facilitate deeper insights about structures and patterns hidden in the graph. In the current EPGM implementation of the *groupBy* operator, a grouping is based on vertex and edge grouping keys (e.g., the type label or property keys) as well as vertex and edge aggregation functions [7]. For temporal grouping, TPGM provides three additional features: First, time-specific value transformation functions (e.g., year or day of week) can be applied to compute time values on the desired granularity for grouping. Second, the *groupBy* operator supports `GROUP BY CUBE` and `GROUP BY ROLLUP` similar to SQL. If this extension is used for vertex or edge grouping keys, the output of the operator is a collection where each graph corresponds to a single combination of the given grouping keys. Third, aggregation on the temporal properties *from* and *to* of the vertices and edges can not only be specified by user-defined functions but by one of the predefined time-specific aggregation functions (e.g., *MinFrom* or *MaxFrom*). For example, having a graph whose edges include time interval definitions, the *AvgDuration* function can be used to determine the average duration of all intervals. The temporal attributes of the super vertices and edges are the minimum and maximum of the grouped elements.

Pattern Matching. Retrieving subgraphs matching a user-defined pattern graph is an important task within the graph analytics domain. In the EPGM a pattern matching operator *query* is already implemented using basic concepts of Neo4j Cypher[2] to define patterns, e.g., `(a)-[b]->(c)` [8]. Predicate functions can be embedded in a pattern inside a `WHERE` clause by using variables

[2] https://neo4j.com/developer/cypher-query-language/

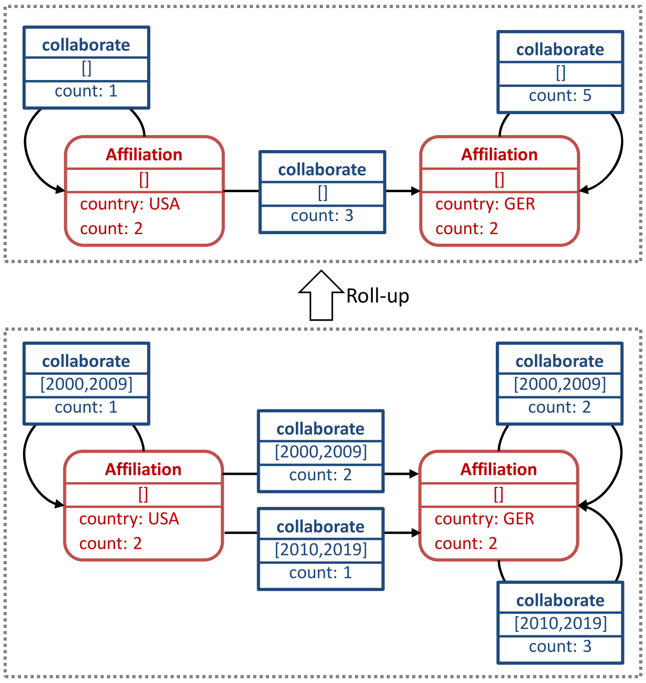

Fig. 4 The result of a time-specific grouping and aggregation applied on the matches of Fig. 3. By using the ROLL UP extension, a collection with two logical graphs is returned

defined in the pattern. Matching subgraphs can be modified by providing a construction pattern. In TPGM, we extend this functionality by using the temporal attributes *from* and *to* inside predicate definitions, i.e., by pre-defined (see Table 2) and user-defined predicate functions. This offers the possibility to define temporal patterns that describe evolutionary behavior of the graph instances. For example, the pattern `(a:Author)-[m:member]->(f:Affiliation country: USA) WHERE m.asOf(2017)` describes an author being a member of an affiliation from the United States as of 2017.

5 Temporal Graph Analytics Workflows

In this section we discuss exemplary workflows for temporal graph analytics. We illustrate how they can be supported by GRADOOP and its extension to TPGM.

Snapshot generation and graph evolution. Graph systems or algorithms might focus on the analysis of static graphs representing the state (or snapshot) of a graph at a specific point in time. GRADOOP therefore supports the retrieval of snapshots using the *snapshot* operator in combination with time-based predicates. To identify the difference and thus the changes between two graph snapshots, the *diff* operator can be used.

Temporal pattern matching. Searching for graph patterns using time-constraints is important for temporal analysis. In the example of Fig. 1, a query to obtain simultaneous collaborations between affiliations must take the valid times of the *member* edges into account:

```
graph.query(
  // Pattern graph
  MATCH
  (f1:Affiliation)
   <-[m1:member]-(a1:Author)
     (a1)-[:write]->(p:Pub)
     (p)<-[:write]-(a2:Author)
     (a2)-[m2:member]->(f2:Affiliation)
  WHERE a1!=a2
  AND m1.overlaps(p) AND m2.overlaps(p),
  // Construction pattern
  (f1)-[:collaborate{from=p.from}]
       ->(f2))
```

By applying this to GRADOOP's *query* operator together with the construction pattern (given as second argument), a collection of matching subgraphs is generated. The result is illustrated in Fig. 3 by means of three collaborations found, which are caused by three publications (*Model Management*, *Citation Analysis* and *CloudFuice*) of the graph in Fig. 1.

Time-specific grouping and aggregation. The time dimension automatically introduces a hierarchy, i.e., graphs can be grouped (summarized) at multiple levels of time-granularity. For example, the graph at the bottom of Fig. 4 summarizes the collaboration between countries per decade based on co-authored publications, i.e., the *Affiliation* vertices are grouped by their *country* property and the *collaborate* edges are aggregated at the level of decades to reflect the temporal changes in the collaboration over time. However, this summarization can be rolled-up on the time hierarchy to have a global aggregation. To this end, GRADOOP's *groupBy* operator can be applied to the graph collection in Fig. 3 to group all *collaborate* edges by year with `GROUP BY ROLLUP` so that the resulting graph collection contains both graphs visualized in Fig. 4. The following snippet illustrates the call of the operator together with its parameters to configure the time-specific grouping and aggregation.

```
graph.groupBy(
 [:label, 'country'],
```

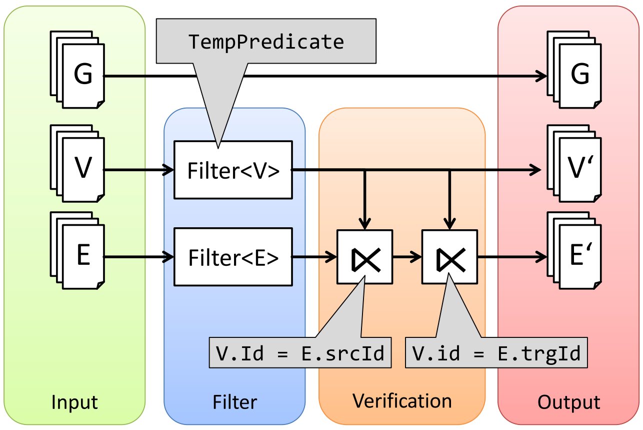

Fig. 5 Distributed dataflow implementation of the snapshot operator, that is realized by two Filter functions. Both consume a temporal predicate function representing the snapshot

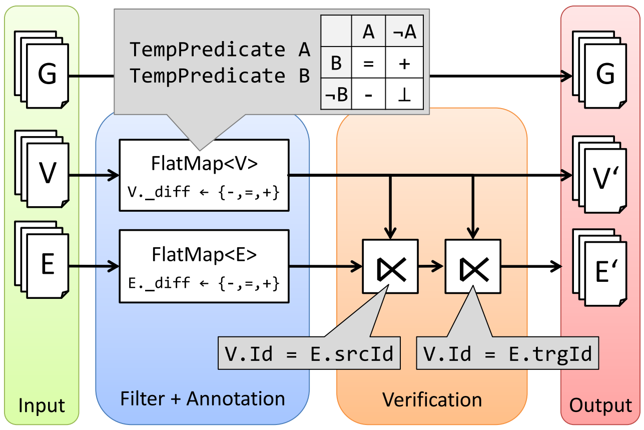

Fig. 6 The diff operator is realized by two FlatMap functions. Both consume temporal predicate functions representing the snapshots

```
[(superVertex['count'] = vertices.
   count())],
[:label, (from, t => DECADE(t))]
   BY ROLLUP,
 [(superEdge['count'] = edges.
        count())])
```

Vertices will be grouped by their respective label and the values of the property *country*. Since the BY ROLLUP extension is specified in addition to the two given edge grouping keys, the grouping results in a graph collection with two logical graphs. In the first one, the edges are grouped by their label and the decade of the from timestamp. Latter is specified by a time-specific value transformation function and represents the beginning of the edge's validity. In the second one, the edges are grouped by their label only. For both vertex and edge aggregation function, a count aggregation is applied.

Efficient maintenance and re-use of queries and analysis results. Repeated execution of GRADOOP workflows, e. g., at certain times (e. g., once a month) or at certain events (e. g., when adding a new publication into a bibliographic network) requires an efficient update of graph transformations and graph summaries both in their structure and aggregated property values. In the above mentioned GROUP BY ROLLUP example, newly added *publication* vertices might only update the *collaborate* edges where the *publication* valid time falls into the valid time of the *collaborate* edge.

6 Implementation

In this section, we provide details of the implementation of two selected TPGM operators (*snapshot* and *diff*) to demonstrate the realization of temporal graph analysis workflows in the distributed in-memory dataflow model. All operation implementations are – as with all other GRADOOP operators – based on Apache Flink. We evaluate both operator implementations in Section 7.

The TPGM graph data model implementation reuses the concepts of the EPGM implementation, which is described by Junghanns et al. in [7]. Thus, a graph is stored in three Flink datasets: one holding vertices (*V*), edges (*E*) and one for graph identifiers (*G*). A *dataset* represents a distributed collection of elements of the same type. The graph identifiers are used to hold information like label and properties of a logical graph instance. Therefore, it contains only one graph identifier whereby a graph collection stores one graph identifier for each graph instance of the collection. To this end, our operator implementations consume three datasets as input and produce three modified datasets as output. The internal logic is composed of different transformations that are applied to these datasets to realize the respective functionality.

Snapshot operator. As described in Section 4, the *snapshot* operator provides the retrieval of a valid snapshot of the entire temporal graph by applying a temporal predicate function, e. g., asOf (see Table 2). Fig. 5 shows the data flow within a simplified architectural sketch. We implemented the *snapshot* operator by using two Flink *filter* transformations: one for the vertices *V* and one for the edges *E* (see the blue box in Fig. 5). Each transformation applies a temporal predicate to each record of *V* and *E*, respectively. The predicate has access to the valid and transaction time of each graph element and thus decides whether to keep the element or to discard it.

Since vertices and edges are handled separately, the filter step may produce dangling edges. These edges are kept by the edge filter but their source and/or destination vertex is

Table 3 LDBC social network dataset statistics including the overhead of disk usage compared to the EPGM

SF	|V|	|E|	Disk usage	Overhead
1	3.3 M	17.9 M	4.3 GB	+34.8%
10	30,4 M	180.4 M	43.1 GB	+44.5%
100	282.6 M	1.77 B	430.0 GB	+45.0%

Table 4 Used predicates for the evaluation, their identifiers and the resulting selectivity

Name	Predicate	Sel. in %
P1	createdIn(2010-01-01,2010-01-20)	0.01
P2	asOf(2010-10-13)	10
P3	asOf(2011-02-18)	20
P4	asOf(2011-06-13)	30

discarded by the vertex filter function. A subsequent verification step is thus performed to remove dangling edges (see the orange box in Fig. 5). The verification is realized by two of Flink's semi-joins: $\ltimes_{V.Id=E.srcId}$ and $\ltimes_{V.Id=E.trgId}$, where *V.Id* is the vertex identifier, *E.srcId* is the identifier of the edge's source vertex and *E.trgId* of the edge's target vertex.

The result of this operator is a valid graph, with $V' \subseteq$ V and $E' \subseteq E$. The graph identifier G remains unchanged.

Difference operator. The *diff* operator can be used to explain the changes in a graph between two snapshots (i.e., states of the graph at a specific time). In our case, a snapshot is represented by a temporal predicate, e.g., `asOf(x)` describes the graph snapshot at time x. The operator extends each graph element by a property that characterized it as added, deleted or persistent.

The architectural sketch of the *diff* operator is illustrated in Fig. 6. Since all temporal information is stored in the input graph, we can apply the two predicates that have to be compared to each other on the same input datasets. Here we exploit a decisive advantage, namely that all graph elements are handled independent of each other and thus can be analyzed independently by both given predicates in a single step. More precisely, having a graph element and two predicates A and B as input, we can check if that element exists in both, none, only the first or only the second snapshot, each represented by the predicates A and B. If it exists in none of the two snapshots, the element will be discarded. Otherwise, it will be annotated with a property `_diff` indicating if the element has been added (`_diff=` +), deleted (–) or left unchanged (=). The upper gray box of Fig. 6 shows the predicate evaluation matrix. The resulting set of (annotated) vertices and edges is thus the union of the vertices and edges of both logical snapshots. We implemented the filtering and annotation step using Flink's `flatMap` transformation operator because `flatMap` takes one element and produces zero, one, or more (modified) elements.

Let us consider the example graph of Fig. 1. The validity of the *Author* vertex named *Rost* begins in 2018 and has an infinite duration. The described operation `graph.diff(asOf(2010),asOf(2018))` evaluates this vertex against both predicates. The first (`asOf(2010)`) would be evaluated to false, the second (`asOf(2018)`) would be evaluated to true. Consequently, this vertex is annotated as "added" (i.e., a property `_diff:+` is added to the vertex) (see Fig. 2).

Analogous to the implementation of the snapshot operator, a subsequent verification step removes possible dangling edges. The result of this operator is again a graph, with $V' \subseteq V$ and $E' \subseteq E$ with the described additional property. Just like the snapshot operator, the graph identifier G remains unchanged.

7 Evaluation

One of the main goals of a distributed shared-nothing system is the ability to respond to growing data sizes or problem complexity by adding resources. We therefore evaluate the scalability of the two selected operator implementations of Section 6 with respect to increasing data volume and computing resources in this section. Since the application of both operators results in a reduction in the number of graph elements, we also evaluate the influence of the filter selectivity on operator scalability.

Setup. The evaluation was performed on a shared-nothing cluster with 16 workers connected via 1 GBit Ethernet. Each worker consists of an Intel Xeon E5-2430 6 x 2.5 Ghz CPU, 48 GB RAM, two 4 TB SATA disks and runs openSUSE 13.2. We use Hadoop 2.6.0 and Flink 1.6.0. We run Flink with 6 threads and 40 GB memory per worker.

We use the LDBC-SNB data set generator [3] to create three datasets with different scale factors (SF). A resulting graph forms a heterogeneous social network with a fixed schema. We extracted the temporal dimension from vertex and edge properties that are holding their creation timestamp to define the valid times. The synthetic graphs represent structural and temporal characteristics of real-world graphs, e.g., successively creation of instances over time and relations as well as skewed property value distributions. Elements are created in the range of the year 2010 to 2012. Table 3 shows some statistics of the three datasets used throughout this evaluation. In addition to the SF used, the cardinality of vertex and edge sets, the dataset size on hard disk as well as the overhead compared to the original EPGM size are specified. Each dataset is stored in

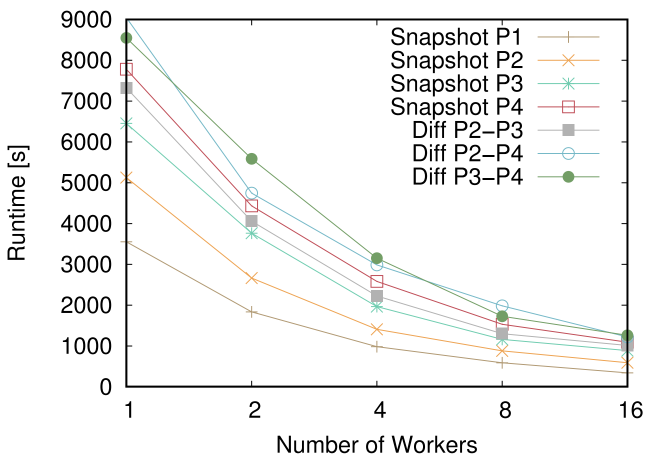

Fig. 7 Runtime of the diff and snapshot operator using predicates P1 through P4 for different numbers of workers. (Dataset LDBC.100)

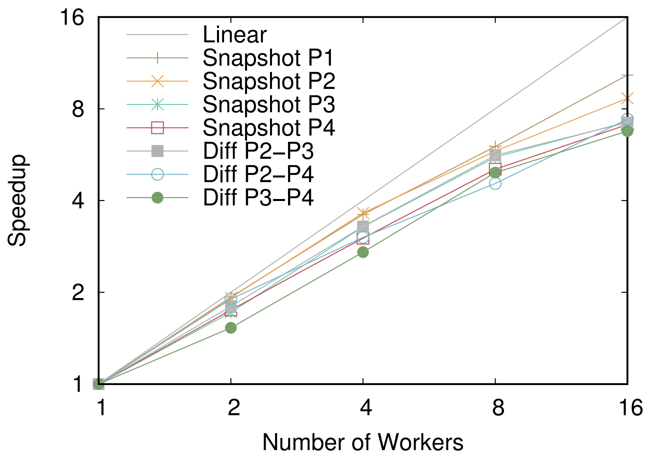

Fig. 8 Speedup of the diff and snapshot operator using predicates P1 through P4 for different numbers of workers. (Dataset LDBC.100)

the Hadoop distributed file system (HDFS). We run three executions per setup and report the average runtimes. Runtimes are measured by Flink's execution environment and include loading the graph from HDFS, executing the operator and writing the resulting graph back to HDFS. In our experiments, we vary the number of workers by setting the parallelism parameter to the respective number of threads (e. g., 2 workers correspond to 12 threads).

Table 4 shows the different predicates used in the experiments. We chose two pre-defined predicate functions (see Table 2) with varying query timestamps which are in the range of the dataset's valid times. The resulting selectivities vary between 0.01% for P1 and 30% for P4. For example, a selectivity of 30% means, that the resulting graph contains about 30% of graph elements compared to the input graph. The selectivities persist for all scaling factors of the datasets.

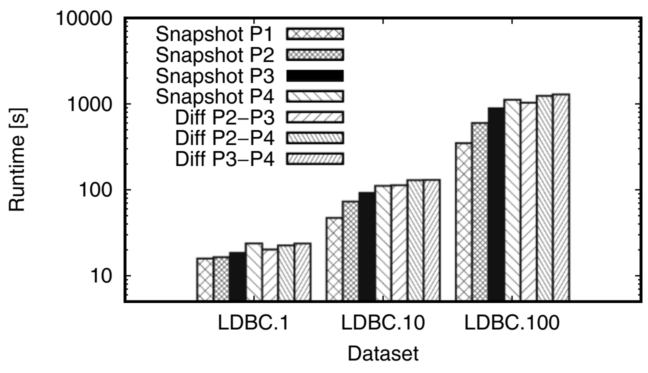

Fig. 9 Runtime of the diff and snapshot operator using predicates P1 through P4 for different datasets. (Numbers of workers = 16)

Experimental results We first evaluate the absolute runtime and relative speedup of our snapshot and diff operator implementation. The operators are executed on each dataset using an increasing number of workers (from 1 to 16) and all predicates of Table 4. The results for the largest dataset LDBC.100 are shown in Fig. 7 and Fig. 8, respectively.

The results show that our Flink-based operator implementations have good scalability. The comparison between snapshot and diff shows that the former has slightly better scalability. This is because filtering within the diff operator generally produces a greater result than filtering within snapshot, because only those elements are removed that do not meet both predicates *A* and *B* (see Fig. 6). This makes the subsequent verification step (semi join) more complex.

For both operators, we are using different predicates to prove that the scalability only slightly depend on the selectivity (see Fig. 8). However, Fig. 7 shows that the running time increases proportionally with selectivity. This can be explained by the fact that the write process part is included in the runtime measurement and that more data has to be written for greater selectivity.

In a second experiment, we varied the datasets for a fixed number of workers (16). Fig. 9 shows the runtimes for both operators using different predicates. We observe an almost linear dependence of the runtime on the size of the respective data sets. For example, diff(P2, P4) takes 128 seconds for LDBC.10 and 1,222 seconds for LDBC.100.

Fig. 10 shows the speedup results for different numbers of workers depending on the selectivity of the predicate used. We observe an almost constant speedup, i. e., the speedup is independent of the predicate's selectivity. Only when using 16 workers, a slight decline in speedup is noticeable but that is probably due to the increased communicative overhead for large number of workers.

Again, through the characteristic of a distributed dataflow engine the duration of a single operator can not be measured, but only the entire workflow. Thus, in the measurement results, it is not possible to distinguish exactly

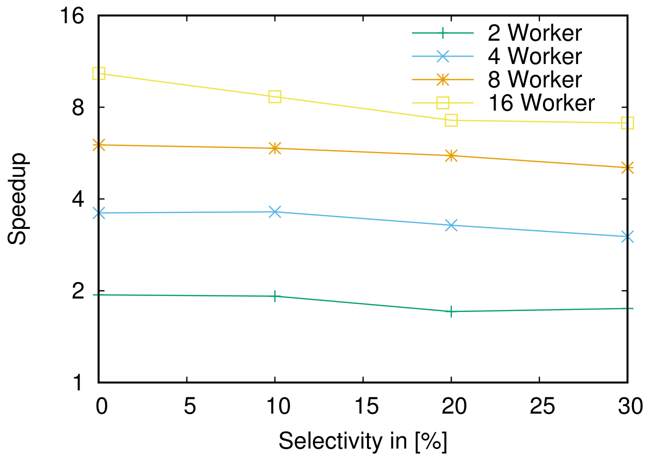

Fig. 10 Relative speedup of the snapshot operator with respect to the selectivity of the predicate used. (Dataset LDBC.100)

between the process of reading the graph, applying the operator and writing the result. Further, different selectivities result in different amounts of data that have to be written, which in turn leads to different time investment for writing.

8 Conclusion

We reported on our work in progress on temporal graph analysis within the distributed graph analytic system GRADOOP. To this end, we introduced the flexible temporal property graph model TPGM that supports bitemporal time semantics. Furthermore, we extended existing and introduced new GRADOOP operators to answer time-respecting analytical questions over evolving graphs. We illustrated the use of these operators within common building blocks of analysis workflows and provided first implementation details and evaluation results that prove a good speedup in a distributed environment.

Temporal graphs and their analysis is an important and promising field of research. In future work we will further extend GRADOOP by temporal features such as operators and algorithms to make GRADOOP a powerful and flexible system for temporal graph analysis.

Acknowledgements This work is partially funded by the German Federal Ministry of Education and Research under grant *BMBF 01IS18026B* and by Sächsische Aufbau Bank (SAB) and the European Regional Development (EFRE) under grant No. *100302179*.

References

1. Cheng R et al (2012) Kineograph: taking the pulse of a fast-changing connected world. Proc EuroSys:85–98. https://doi.org/10.1145/2168836.2168846
2. Date CJ, Darwen H, Lorentzos N (2002) Temporal data & the relational model. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA
3. Erling O, Averbuch A, Larriba-Pey J, Chafi H, Gubichev A, Prat A, Pham MD, Boncz P (2015) The LDBC social network benchmark: Interactive workload. In: Proceedings of the 2015 ACM SIGMOD international conference on management of data. ACM, New York, pp 619–630. https://doi.org/10.1145/2723372.2742786
4. Holme P, Saramäki J (2012) Temporal networks. Phys Rep 519(3):97–125. https://doi.org/10.1016/j.physrep.2012.03.001
5. Junghanns M, Kießling M, Teichmann N, Gómez K, Petermann A, Rahm E (2018) Declarative and distributed graph analytics with GRADOOP. Proc VLDB Endow 11(12):2006–2009
6. Junghanns M, Petermann A, Neumann M, Rahm E (2017) Management and analysis of big graph data: current systems and open challenges. In: Handbook of big data technologies. Springer, Cham, pp 457–505
7. Junghanns M, Petermann A, Rahm E (2017) Distributed grouping of property graphs with GRADOOP. Proc BTW, P-265:103–122
8. Junghanns M, Petermann A, Teichmann N, Gómez K, Rahm E (2016) Analyzing extended property graphs with Apache Flink. In: Proc. SIGMOD Workshop on Network Data Analytics
9. Khurana U, Deshpande A (2013) Efficient snapshot retrieval over historical graph data. Proc ICDE, 997–1008. https://doi.org/10.1109/icde.2013.6544892
10. Kulkarni K, Michels J (2012) Temporal features in SQL: 2011. SIGMOD Rec 41(3):34–43
11. Ligtenberg W, Pei Y, Fletcher G, Pechenizkiy M (2018) Tink: A temporal graph analytics library for Apache Flink. In: WWW '18 Companion Proceedings of the The Web Conference 2018, pp 71–72. https://doi.org/10.1145/3184558.3186934
12. Miao Y et al (2015) Immortalgraph: a system for storage and analysis of temporal graphs. ACM Trans Storage 11(3):14
13. Pigné Y, Dutot A, Guinand F, Olivier D (2008) Graphstream: A tool for bridging the gap between complex systems and dynamic graphs. CoRR
14. Rost C, Thor A, Rahm E (2019) Temporal graph analysis using gradoop. In: BTW 2019 - Workshopband. Lecture Notes in Informatics (LNI), vol P-290. Gesellschaft für Informatik, Bonn, pp 109–118
15. Steer BA, Cuadrado F, Clegg RG (2020) Raphtory: Streaming analysis of distributed temporal graphs. Future Generation Computer Systems 102:453–464. https://doi.org/10.1016/j.future.2019.08.022
16. Then M, Kersten T, Günnemann S, Kemper A, Neumann T (2017) Automatic algorithm transformation for efficient multi-snapshot analytics on temporal graphs. Proc VLDB Endow 10(8):877–888