# CONTENTS

# Exploration and Analysis of Temporal Property Graphs

Christopher Rost
University of Leipzig & ScaDS.AI
Dresden/Leipzig, Germany
rost@informatik.uni-leipzig.de

Kevin Gomez
University of Leipzig & ScaDS.AI
Dresden/Leipzig, Germany
gomez@informatik.uni-leipzig.de

Philip Fritzsche
University of Leipzig & ScaDS.AI
Dresden/Leipzig, Germany
fritzsche@informatik.uni-leipzig.de

Andreas Thor
Leipzig University of Applied Sciences, Germany
andreas.thor@htwk-leipzig.de

Erhard Rahm
University of Leipzig & ScaDS.AI
Dresden/Leipzig, Germany
rahm@informatik.uni-leipzig.de

## ABSTRACT

We demonstrate the *Temporal Graph Explorer*, a distributed open-source framework that enables time-dependent graph exploration and analysis on large real-world networks using a rich temporal property graph model and dynamic graph operators. In this demonstration we show how the evolution of a graph plays an essential role in many analytical questions. Besides retrieving a snapshot from a past graph state or calculating the difference between two graph snapshots, users can use our application to summarize the graph to reduce its complexity and to obtain deeper insights into its evolution. Visitors of the demo can visually experience these advanced temporal operators and their results or build and manipulate analytical programs in a declarative way without extended programming skills and run them on a distributed local or remote environment. We provide real-world temporal graph data from bicycle rentals of New York City with millions of rentals per month, also demonstrating horizontal scalability of the system.

## 1 INTRODUCTION

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

In this paper, we demonstrate the *Temporal Graph Explorer*, a tool for user-friendly exploration, analysis, and visualization of large temporal property graphs. The core of this application is Gradoop[1], an open-source framework for distributed graph analysis. Its *Temporal Property Graph Model (TPGM)* [8] enables modeling and analysis of graphs with bitemporal time semantics. The TPGM also comes with a set of composable temporal graph operators (including snapshot retrieval, graph evolution, and time-dependent grouping and aggregation) that can be visually configured through the user interface or programmatically

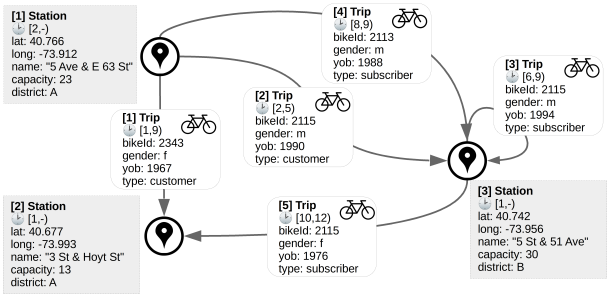


**Figure 1: Example temporal property graph representing bicycle rentals between rental stations. The validity period of an edge is marked with a clock symbol and simplified with numbers instead of timestamps.**

combined with the help of the declarative language GrALa [5] to build distributed analysis workflows.

The *Temporal Graph Explorer* thus enables the analysis of the evolution of graphs, i.e., to figure out *when* something happened or changed, rather than a static view representing something that happened at some time [9]. To this end, it provides temporal graph operators (including **snapshot** retrieval and graph **difference**) to compute and visualize changes, including additions and deletions, that have been occurred. This can be used in the given bicycle example to find, for a given week in the past, all rentals that have been added, removed, or remained the same.

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

## 2 TEMPORAL GRAPH ANALYSIS AND EXPLORATION

The *Temporal Graph Explorer* is an application to explore, analyze and visualize temporal property graphs. An intuitive web-based user interface enables the configuration of selected temporal

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

© 2021 Copyright held by the owner/author(s). Published in Proceedings of the 24th International Conference on Extending Database Technology (EDBT), March 23-26, 2021, ISBN 978-3-89318-084-4 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

graph operators and their application on a predefined graph dataset of the user's choice. The whole processing is then executed in a distributed environment by using Gradoop as a backend framework. The resulting graph is again temporal and sent back to the user interface for visualization. Frontend and backend application communicate through a RESTful webservice. The visual representation is adjusted to the temporal characteristics of the graph, as later explained. An architectural overview of the system is given in Figure 2.

In the following, we will dive into the single parts of the application including the used bitemporal graph data model, three selected operators, and graph visualization techniques focused on understanding the graph's evolution.

***Graph model and distributed processing.*** The heart of our *Temporal Graph Explorer* is Gradoop [4, 5], which offers many generic operators on graphs (for pattern matching, global aggregations, etc.) that can be used within workflows for graph analysis. Workflows representing graph analytical programs can be expressed in GrALa for distributed execution. All operators are closed over the model, i.e., they take graphs as input and produce graphs. Since Gradoop is built on top of Apache Flink's Dataset-API, each operator is based on a subset of Flink's transformations (map, flatmap, join, etc.) to achieve a parallel execution and scalability to large graphs. Thus, it combines and extends features of graph analytical systems with the benefits of distributed graph processing.

To maintain the full history of a graph, including any insertion, deletion, or update of a vertex, edge, or its properties, Gradoop was extended by the recently introduced TPGM [7, 8] as a graph data model. It supports labeled, directed, multi-graphs where the vertices and edges are characterized by a unique identifier, a type label, and a set of properties, modeled as key-value pairs. In addition, each vertex and edge is extended by two time intervals $I_v$ and $I_{tx}$ that define a lifespan regarding to valid-[2] and transaction time dimension, which ensures a bitemporal data modeling [3]. Each close-open interval is represented by two first-order attributes $t_{\text{from}}$ and $t_{\text{to}}$ defining the start and end timestamp of the respective time interval. Thus, a graph of this model contains all historical and rollback information and therefore allows the exploration of its evolution and retrieving valid snapshots from the past, present or future for the valid time dimension or past and present states from the transaction time domain. To provide maximum flexibility, $I_v$ can also be empty if no time information is available or just hold a single timestamp $t_{\text{from}}$ if the duration of the element is not available or can be neglected, e.g., the time at which a bike station was newly built. A more detailed description and scalability evaluation of the TPGM and its operators is given in [8].

***Graph snapshot and difference.*** To enable temporal and evolutionary queries and analysis, one data management challenge for large historical graphs is the retrieval of graph snapshots as of any time-point in the considered time domain [6]. To achieve this, we developed the *snapshot* operator that can be applied on a temporal graph instance and allows to retrieve a valid state of the graph either at a specific point in time or a subgraph that is valid during a time range. The user can configure the operator by pre-defined time-dependent predicates such as *asOf()*, *overlaps()* or *precedes()*. Furthermore, user-defined predicates, as well as helper functions to extract certain time dimensions, can be used.

[2]Valid-time is also known as application time.

An important part of the analysis of graphs is the examination of changes that have occurred between two points in time. Changes, i.e. additions, deletions, and edits, represent the evolution of a temporal graph and can be selected or aggregated in subsequent analysis steps. Therefore, we demonstrate the *difference* operator that computes a graph $\triangle G$ between two graph snapshots $G_1$ and $G_2$ by determining the union $G_1 \cup G_2$ and extending each element by a property that expresses the addition, deletion, or persistence of this element respectively. The user configures both snapshots by using time-dependent predicate functions, as described before. In addition, the desired time-dimension can be selected.

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

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

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

***Analysis specification and result visualization.*** As already mentioned, Gradoop workflows are described in the declarative language GrALa. Besides the possibility for users to write programs directly in GrALa (see programmatic demonstration) the Temporal Graph Explorer allows the creation of simple workflows with the help of an adaptive user interface. Users can select operator(s) and the user interface allows easy parameterization by displaying suitable graphical elements, e.g. drop-down lists for selecting a snapshot predicate.

The resulting temporal graph is visualized by the Temporal Graph Explorer using Apache ECharts[3] and the JavaScript library of Cytoscape[4], an open-source software platform for visualizing

[3]https://echarts.apache.org
[4]https://js.cytoscape.org/

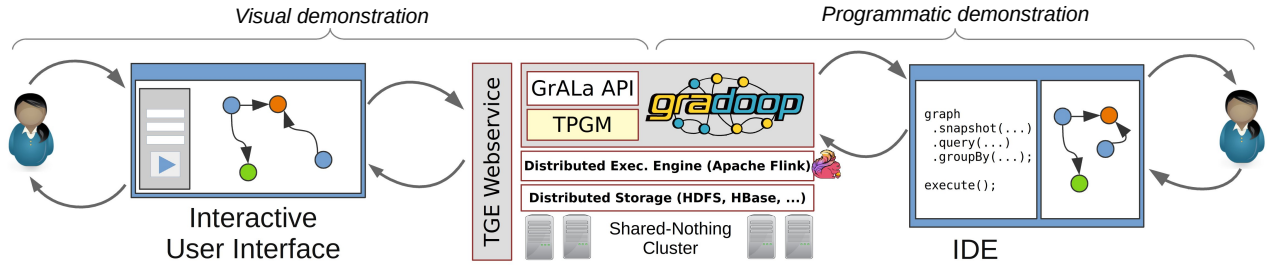


**Figure 2: System architecture overview of Temporal Graph Explorer and user interaction workflow for both demonstration scenarios.**

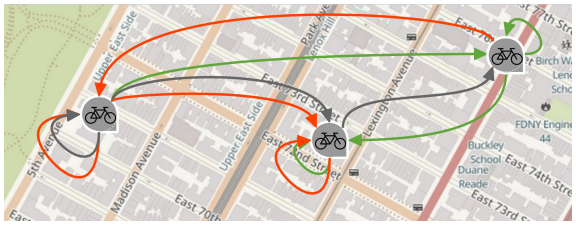

**Figure 3: Example visualization of a difference graph.**

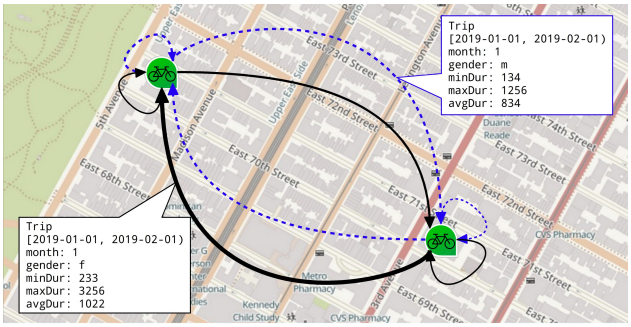


**Figure 4: Visualization of two grouped stations each placed in the grid center and their aggregated trips distinguished by gender (blue dotted = male; black solid = female).**

complex attributed networks. By default, the coloration of vertices and edges is based on the respective label, i.e., elements with the same label are equally colored. Further, identifiers, properties, and temporal attributes are displayed in a tooltip after selecting a vertex or an edge.

Besides this default graph visualization, we provide two additional graph representations: a difference graph view and a grouped graph view. Figure 3 shows a cutout of a difference graph's visualization. It colorizes the vertices and edges depending on the annotation with which the elements are expanded by the difference operator. A vertex or edge is colored red if the elements have been deleted in the time between both snapshots, grey if the elements were not changed at all, or green if the elements were created.

For the visualized result of the grouping operator, aggregated properties (e.g., `count`, `minDuration`, `maxTimestamp`) can be used to adjust the radius of vertices and width of edges to the corresponding property value. For example, the width of a super-edge depends on the average duration of the grouped edges. Besides, if the graph data set contains geographic coordinates as properties of vertices, these can be mapped onto an interactive map using a geo-layout (see the example in Section 3).

Further, the graph can be exported to the graph description language *DOT*, which can be easily rendered by *Graphviz* library or *Gephi*, an external visualization and exploration tool.

## 3 DEMONSTRATION DESCRIPTION

Our demonstration of the *Temporal Graph Explorer* is separated into two parts as shown in Figure 2. First, we demonstrate our web-based toolbox and user-interface for Gradoop. There, different analytical operators can be selected and configured in many ways, whereas the resulting graph is presented to the user by graph visualization to present the analytical value of the operators. We provide real-world graph data based on the open-source New York CitiBike[5] database which captures bicycle rentals since the year 2013. The structure of the graph is analogous to the example graph of Figure 1. The dataset consists of around 1,000 vertices (ascending over time) representing the rental stations and more than 2,000 trip edges per hour, which results in about 17 million trips per year. Further, we also integrated other temporal networks such as a heterogeneous social network synthetically generated by the LDBC data generator [2]. In the second part of the demonstration, we give visitors the opportunity to inspect and manipulate existing example temporal programs written in GrALa. All our examples can be executed on demonstration data locally, but also remotely on our research cluster.

***Visual demonstration.*** To give more insight about results of frequently used single temporal operators, such as *snapshot*, *difference* and *temporal grouping*, we demonstrate their usage by the *Temporal Graph Explorer*. The explorer offers each visitor the possibility to parameterize the operators by using a dynamic user interface. Each operator is then executed by a Gradoop instance from where resulting temporal graphs are pushed back to the web application for visualization. The visualized graph is interactive, i.e., visitors can zoom in and out, drag vertices and edges to other positions or click on them to show their properties and temporal attributes.

To demonstrate the *snapshot* operator, a user is able to choose between supported time predicates, e.g., *asOf*, *fromTo* or *overlaps*, a time dimension (valid- or transaction time) and a respective timestamp or interval. Through the visualization, the user receives instant feedback on the changes made and thus can explore and compare various states of different times.

For the *difference* operator, a user can compare two temporal graph snapshots by exploring a difference graph. To define the

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

```
1  bikeGraph
2   // get all elements that overlap year 2019
3   .snapshot(
4    new Overlaps(
5     LocalDateTime.of(2019, 1, 1, 0, 0),
6     LocalDateTime.of(2019, 12, 31, 23, 59)))
7   // remove dangling edges
8   .verify()
9   // filter edges by year of birth
10  .subgraph(e -> e.getProperty("yob") > 1990)
11  // summarize the graph
12  .groupBy(
13   // group vertices by label and grid id
14   [
15    label(),
16    v -> getGridId(v)
17   ],
18   // do not aggregate vertices
19   [ ],
20   // group edges by label, month and gender property
21   [
22    label(),
23    timeStamp(VALID_TIME, FROM, ChronoField.MONTH_OF_YEAR),
24    property("gender")
25   ],
26   // calc min, max and avg duration for grouped edges
27   [
28    new MinDuration("minDur", VALID_TIME),
29    new MaxDuration("maxDur", VALID_TIME),
30    new AverageDuration("avgDur", VALID_TIME)
31   ]
32  )
```

**Figure 5: Analytical application defined in GrALa.**

snapshots, we are using the same implemented time predicates. For the visualization, graph elements are colorized to provide more information about their temporal evolution as described in Section 2. Using this kind of visualization a user gains insights about how and how frequently the graph evolves between two graph states.

*Temporal grouping*, the last operator we demonstrate, offers a large variety of possibilities to explore the temporal graph by summarizing it. The configuration options of this operator are very extensive and depend on the characteristics of the selected graph dataset and the objectives of the corresponding analysis. The *Temporal Graph Explorer* supports the user in the configuration of the operator by a flexible selection of the predefined key functions for vertices and edges, as well as aggregate functions. Appropriate arguments are offered for parameterized key functions. For example, a list of property names is offered for the function `property(<name>)`. Timestamps which appear to be useful for the selected temporal graph are also suggested for use with temporal key functions. Besides, the user can choose from pre-defined aggregate functions to additionally configure the grouping and thus to enrich the grouped elements with detailed information about the grouped element, which can be accessed in the graph visualization. The user can thus interactively add key and aggregate functions to the configuration step by step until the grouped graph and its aggregated values provide information about a specific analysis question. Figure 4 shows a cutout of a grouping result in the *Temporal Graph Explorer*. The used configuration is equal to the later-described analytical application. Since the grouped vertices have geographic properties, they can be placed on a map-view. Edges can also be colored according to a certain property, as one can see in the figure. The properties created by the aggregates are displayed after selecting a vertex or an edge.

***Programmatic demonstration.*** For a better understanding of the API of our temporal operators, we prepared a set of example programs[6] that show basic functionality and usage. Advanced examples, like the one in Figure 5, demonstrate the composition of multiple (temporal-) operators to answer specific analytical questions.

In an example scenario, we want to answer the following question: *In 2019, how did the minimum, maximum and average trip duration change per month for male and female users born after 1990 between stations located in different areas?*

We first use the *snapshot* operator (line 3) to retrieve all information about trips, users, and stations of the whole year of 2019. Since *snapshot* can produce dangling edges, we make sure to remove them by calling the *verify* operator (line 8). We further apply a *subgraph* operator with specific predicates (line 10) to filter for users born after 1990. At the end of our pipeline, we want to summarize our graph by calling the *grouping* operator with specific grouping key functions for vertices (lines 15-16) and edges (lines 22-24). Besides the predefined *label()* function, we also show the usage of a user-defined grouping key function (line 16). It calculates a map grid using latitude and longitude properties of our vertices. We further group the edges for each month of the year, separated by the two genders of the users. During this step, we also apply multiple aggregation functions to calculate the minimum, maximum, and average duration of trips (lines 28-30). Figure 4 shows the results of this GrALa program for two randomly picked stations.

The example illustrates the level of abstraction using our operators. A user does not have to care about the underlying graph data structure, operator implementation, or distributed execution details. Gradoop offers a variety of *DataSources* and *DataSinks* to read and write different kinds of data such as *.csv* files for data exchange or *.dot* files for graph visualization.

A visitor of this demonstration has the possibility to freely manipulate the provided GrALa programs. For example, it is possible to use *asOf* instead of *overlaps* at the *snapshot* operator (line 3) to specify a different time period. Further a user is able to use *pattern matching* instead of *subgraph* (line 10) to detect important graph patterns. Also the *grouping* operator can be parameterized by different pre- or user-defined functions to summarize different aspects of the data.

## ACKNOWLEDGEMENT

This work is partially funded by the German Federal Ministry of Education and Research under grant `BMBF 01IS18026B`.

## REFERENCES

[1] Charu Aggarwal and Karthik Subbian. 2014. Evolutionary network analysis: A survey. *ACM CSUR* 47, 1 (2014), 1–36.
[2] Orri Erling et al. 2015. The LDBC social network benchmark: Interactive workload. In *Proc. SIGMOD*.
[3] Tom Johnston. 2014. *Bitemporal data: theory and practice.* Newnes.
[4] Martin Junghanns et al. 2016. Analyzing Extended Property Graphs with Apache Flink. In *Proc. SIGMOD NDA Workshop*.
[5] Martin Junghanns, Max Kießling, Niklas Teichmann, Kevin Gómez, André Petermann, and Erhard Rahm. 2018. Declarative and distributed graph analytics with Gradoop. *PVLDB* 11, 12 (2018), 2006–2009.
[6] Udayan Khurana and Amol Deshpande. 2013. Efficient snapshot retrieval over historical graph data. In *Proc. ICDE*. IEEE, 997–1008.
[7] Christopher Rost, Andreas Thor, Philip Fritzsche, Kevin Gómez, and Erhard Rahm. 2019. Evolution Analysis of Large Graphs with Gradoop. In *Proc. ECML PKDD*. Springer, 402–408.
[8] Christopher Rost, Andreas Thor, and Erhard Rahm. 2019. Analyzing Temporal Graphs with Gradoop. *Datenbank-Spektrum* 19, 3 (2019), 199–208.
[9] Yishu Wang, Ye Yuan, Yuliang Ma, and Guoren Wang. 2019. Time-dependent graphs: Definitions, applications, and algorithms. *Data Science and Engineering* 4, 4 (2019), 352–366.

[6] https://git.io/JvSu0

# CONTENTS

# Exploration and Analysis of Temporal Property Graphs

Christopher Rost
University of Leipzig & ScaDS.AI
Dresden/Leipzig, Germany
rost@informatik.uni-leipzig.de

Kevin Gomez
University of Leipzig & ScaDS.AI
Dresden/Leipzig, Germany
gomez@informatik.uni-leipzig.de

Philip Fritzsche
University of Leipzig & ScaDS.AI
Dresden/Leipzig, Germany
fritzsche@informatik.uni-leipzig.de

Andreas Thor
Leipzig University of Applied Sciences, Germany
andreas.thor@htwk-leipzig.de

Erhard Rahm
University of Leipzig & ScaDS.AI
Dresden/Leipzig, Germany
rahm@informatik.uni-leipzig.de

## ABSTRACT

We demonstrate the *Temporal Graph Explorer*, a distributed open-source framework that enables time-dependent graph exploration and analysis on large real-world networks using a rich temporal property graph model and dynamic graph operators. In this demonstration we show how the evolution of a graph plays an essential role in many analytical questions. Besides retrieving a snapshot from a past graph state or calculating the difference between two graph snapshots, users can use our application to summarize the graph to reduce its complexity and to obtain deeper insights into its evolution. Visitors of the demo can visually experience these advanced temporal operators and their results or build and manipulate analytical programs in a declarative way without extended programming skills and run them on a distributed local or remote environment. We provide real-world temporal graph data from bicycle rentals of New York City with millions of rentals per month, also demonstrating horizontal scalability of the system.

## 1 INTRODUCTION

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

In this paper, we demonstrate the *Temporal Graph Explorer*, a tool for user-friendly exploration, analysis, and visualization of large temporal property graphs. The core of this application is Gradoop[1], an open-source framework for distributed graph analysis. Its *Temporal Property Graph Model (TPGM)* [8] enables modeling and analysis of graphs with bitemporal time semantics. The TPGM also comes with a set of composable temporal graph operators (including snapshot retrieval, graph evolution, and time-dependent grouping and aggregation) that can be visually configured through the user interface or programmatically

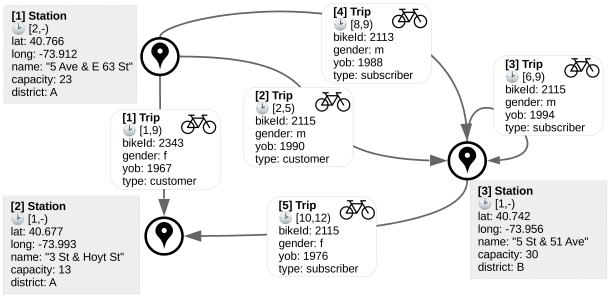


**Figure 1: Example temporal property graph representing bicycle rentals between rental stations. The validity period of an edge is marked with a clock symbol and simplified with numbers instead of timestamps.**

combined with the help of the declarative language GrALa [5] to build distributed analysis workflows.

The *Temporal Graph Explorer* thus enables the analysis of the evolution of graphs, i.e., to figure out *when* something happened or changed, rather than a static view representing something that happened at some time [9]. To this end, it provides temporal graph operators (including **snapshot** retrieval and graph **difference**) to compute and visualize changes, including additions and deletions, that have been occurred. This can be used in the given bicycle example to find, for a given week in the past, all rentals that have been added, removed, or remained the same.

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

## 2 TEMPORAL GRAPH ANALYSIS AND EXPLORATION

The *Temporal Graph Explorer* is an application to explore, analyze and visualize temporal property graphs. An intuitive web-based user interface enables the configuration of selected temporal

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

© 2021 Copyright held by the owner/author(s). Published in Proceedings of the 24th International Conference on Extending Database Technology (EDBT), March 23-26, 2021, ISBN 978-3-89318-084-4 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

graph operators and their application on a predefined graph dataset of the user's choice. The whole processing is then executed in a distributed environment by using Gradoop as a backend framework. The resulting graph is again temporal and sent back to the user interface for visualization. Frontend and backend application communicate through a RESTful webservice. The visual representation is adjusted to the temporal characteristics of the graph, as later explained. An architectural overview of the system is given in Figure 2.

In the following, we will dive into the single parts of the application including the used bitemporal graph data model, three selected operators, and graph visualization techniques focused on understanding the graph's evolution.

***Graph model and distributed processing.*** The heart of our *Temporal Graph Explorer* is Gradoop [4, 5], which offers many generic operators on graphs (for pattern matching, global aggregations, etc.) that can be used within workflows for graph analysis. Workflows representing graph analytical programs can be expressed in GrALa for distributed execution. All operators are closed over the model, i.e., they take graphs as input and produce graphs. Since Gradoop is built on top of Apache Flink's Dataset-API, each operator is based on a subset of Flink's transformations (map, flatmap, join, etc.) to achieve a parallel execution and scalability to large graphs. Thus, it combines and extends features of graph analytical systems with the benefits of distributed graph processing.

To maintain the full history of a graph, including any insertion, deletion, or update of a vertex, edge, or its properties, Gradoop was extended by the recently introduced TPGM [7, 8] as a graph data model. It supports labeled, directed, multi-graphs where the vertices and edges are characterized by a unique identifier, a type label, and a set of properties, modeled as key-value pairs. In addition, each vertex and edge is extended by two time intervals $I_v$ and $I_{tx}$ that define a lifespan regarding to valid-[2] and transaction time dimension, which ensures a bitemporal data modeling [3]. Each close-open interval is represented by two first-order attributes $t_{\text{from}}$ and $t_{\text{to}}$ defining the start and end timestamp of the respective time interval. Thus, a graph of this model contains all historical and rollback information and therefore allows the exploration of its evolution and retrieving valid snapshots from the past, present or future for the valid time dimension or past and present states from the transaction time domain. To provide maximum flexibility, $I_v$ can also be empty if no time information is available or just hold a single timestamp $t_{\text{from}}$ if the duration of the element is not available or can be neglected, e.g., the time at which a bike station was newly built. A more detailed description and scalability evaluation of the TPGM and its operators is given in [8].

***Graph snapshot and difference.*** To enable temporal and evolutionary queries and analysis, one data management challenge for large historical graphs is the retrieval of graph snapshots as of any time-point in the considered time domain [6]. To achieve this, we developed the *snapshot* operator that can be applied on a temporal graph instance and allows to retrieve a valid state of the graph either at a specific point in time or a subgraph that is valid during a time range. The user can configure the operator by pre-defined time-dependent predicates such as *asOf()*, *overlaps()* or *precedes()*. Furthermore, user-defined predicates, as well as helper functions to extract certain time dimensions, can be used.

[2]Valid-time is also known as application time.

An important part of the analysis of graphs is the examination of changes that have occurred between two points in time. Changes, i.e. additions, deletions, and edits, represent the evolution of a temporal graph and can be selected or aggregated in subsequent analysis steps. Therefore, we demonstrate the *difference* operator that computes a graph $\triangle G$ between two graph snapshots $G_1$ and $G_2$ by determining the union $G_1 \cup G_2$ and extending each element by a property that expresses the addition, deletion, or persistence of this element respectively. The user configures both snapshots by using time-dependent predicate functions, as described before. In addition, the desired time-dimension can be selected.

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

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

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

***Analysis specification and result visualization.*** As already mentioned, Gradoop workflows are described in the declarative language GrALa. Besides the possibility for users to write programs directly in GrALa (see programmatic demonstration) the Temporal Graph Explorer allows the creation of simple workflows with the help of an adaptive user interface. Users can select operator(s) and the user interface allows easy parameterization by displaying suitable graphical elements, e.g. drop-down lists for selecting a snapshot predicate.

The resulting temporal graph is visualized by the Temporal Graph Explorer using Apache ECharts[3] and the JavaScript library of Cytoscape[4], an open-source software platform for visualizing

[3]https://echarts.apache.org
[4]https://js.cytoscape.org/

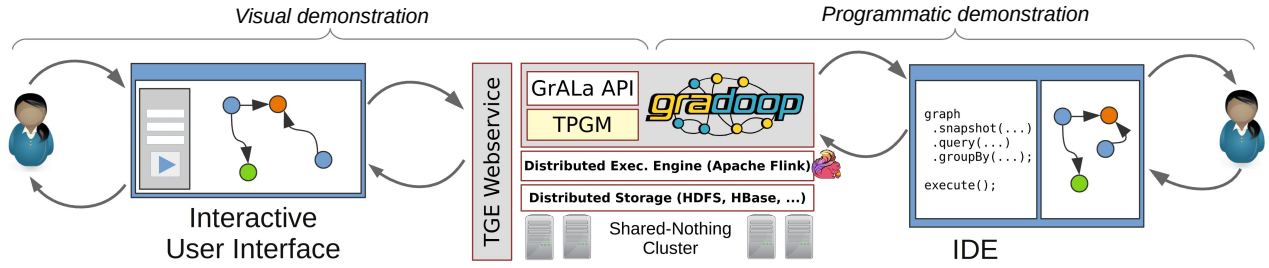


**Figure 2: System architecture overview of Temporal Graph Explorer and user interaction workflow for both demonstration scenarios.**

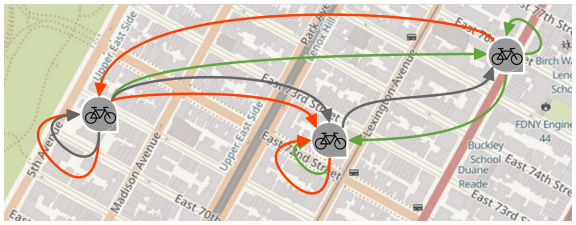

**Figure 3: Example visualization of a difference graph.**

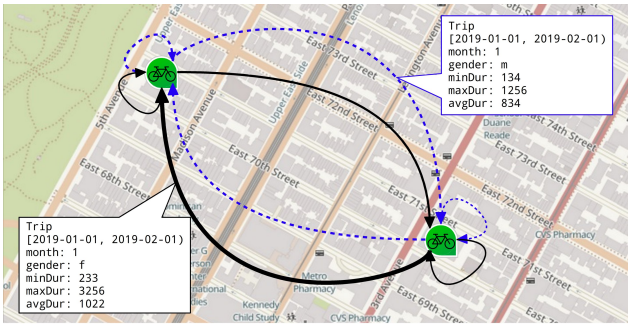


**Figure 4: Visualization of two grouped stations each placed in the grid center and their aggregated trips distinguished by gender (blue dotted = male; black solid = female).**

complex attributed networks. By default, the coloration of vertices and edges is based on the respective label, i.e., elements with the same label are equally colored. Further, identifiers, properties, and temporal attributes are displayed in a tooltip after selecting a vertex or an edge.

Besides this default graph visualization, we provide two additional graph representations: a difference graph view and a grouped graph view. Figure 3 shows a cutout of a difference graph's visualization. It colorizes the vertices and edges depending on the annotation with which the elements are expanded by the difference operator. A vertex or edge is colored red if the elements have been deleted in the time between both snapshots, grey if the elements were not changed at all, or green if the elements were created.

For the visualized result of the grouping operator, aggregated properties (e.g., `count`, `minDuration`, `maxTimestamp`) can be used to adjust the radius of vertices and width of edges to the corresponding property value. For example, the width of a super-edge depends on the average duration of the grouped edges. Besides, if the graph data set contains geographic coordinates as properties of vertices, these can be mapped onto an interactive map using a geo-layout (see the example in Section 3).

Further, the graph can be exported to the graph description language *DOT*, which can be easily rendered by *Graphviz* library or *Gephi*, an external visualization and exploration tool.

## 3 DEMONSTRATION DESCRIPTION

Our demonstration of the *Temporal Graph Explorer* is separated into two parts as shown in Figure 2. First, we demonstrate our web-based toolbox and user-interface for Gradoop. There, different analytical operators can be selected and configured in many ways, whereas the resulting graph is presented to the user by graph visualization to present the analytical value of the operators. We provide real-world graph data based on the open-source New York CitiBike[5] database which captures bicycle rentals since the year 2013. The structure of the graph is analogous to the example graph of Figure 1. The dataset consists of around 1,000 vertices (ascending over time) representing the rental stations and more than 2,000 trip edges per hour, which results in about 17 million trips per year. Further, we also integrated other temporal networks such as a heterogeneous social network synthetically generated by the LDBC data generator [2]. In the second part of the demonstration, we give visitors the opportunity to inspect and manipulate existing example temporal programs written in GrALa. All our examples can be executed on demonstration data locally, but also remotely on our research cluster.

***Visual demonstration.*** To give more insight about results of frequently used single temporal operators, such as *snapshot*, *difference* and *temporal grouping*, we demonstrate their usage by the *Temporal Graph Explorer*. The explorer offers each visitor the possibility to parameterize the operators by using a dynamic user interface. Each operator is then executed by a Gradoop instance from where resulting temporal graphs are pushed back to the web application for visualization. The visualized graph is interactive, i.e., visitors can zoom in and out, drag vertices and edges to other positions or click on them to show their properties and temporal attributes.

To demonstrate the *snapshot* operator, a user is able to choose between supported time predicates, e.g., *asOf*, *fromTo* or *overlaps*, a time dimension (valid- or transaction time) and a respective timestamp or interval. Through the visualization, the user receives instant feedback on the changes made and thus can explore and compare various states of different times.

For the *difference* operator, a user can compare two temporal graph snapshots by exploring a difference graph. To define the

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

```
1  bikeGraph
2   // get all elements that overlap year 2019
3   .snapshot(
4    new Overlaps(
5     LocalDateTime.of(2019, 1, 1, 0, 0),
6     LocalDateTime.of(2019, 12, 31, 23, 59)))
7   // remove dangling edges
8   .verify()
9   // filter edges by year of birth
10  .subgraph(e -> e.getProperty("yob") > 1990)
11  // summarize the graph
12  .groupBy(
13   // group vertices by label and grid id
14   [
15    label(),
16    v -> getGridId(v)
17   ],
18   // do not aggregate vertices
19   [ ],
20   // group edges by label, month and gender property
21   [
22    label(),
23    timeStamp(VALID_TIME, FROM, ChronoField.MONTH_OF_YEAR),
24    property("gender")
25   ],
26   // calc min, max and avg duration for grouped edges
27   [
28    new MinDuration("minDur", VALID_TIME),
29    new MaxDuration("maxDur", VALID_TIME),
30    new AverageDuration("avgDur", VALID_TIME)
31   ]
32  )
```

**Figure 5: Analytical application defined in GrALa.**

snapshots, we are using the same implemented time predicates. For the visualization, graph elements are colorized to provide more information about their temporal evolution as described in Section 2. Using this kind of visualization a user gains insights about how and how frequently the graph evolves between two graph states.

*Temporal grouping*, the last operator we demonstrate, offers a large variety of possibilities to explore the temporal graph by summarizing it. The configuration options of this operator are very extensive and depend on the characteristics of the selected graph dataset and the objectives of the corresponding analysis. The *Temporal Graph Explorer* supports the user in the configuration of the operator by a flexible selection of the predefined key functions for vertices and edges, as well as aggregate functions. Appropriate arguments are offered for parameterized key functions. For example, a list of property names is offered for the function `property(<name>)`. Timestamps which appear to be useful for the selected temporal graph are also suggested for use with temporal key functions. Besides, the user can choose from pre-defined aggregate functions to additionally configure the grouping and thus to enrich the grouped elements with detailed information about the grouped element, which can be accessed in the graph visualization. The user can thus interactively add key and aggregate functions to the configuration step by step until the grouped graph and its aggregated values provide information about a specific analysis question. Figure 4 shows a cutout of a grouping result in the *Temporal Graph Explorer*. The used configuration is equal to the later-described analytical application. Since the grouped vertices have geographic properties, they can be placed on a map-view. Edges can also be colored according to a certain property, as one can see in the figure. The properties created by the aggregates are displayed after selecting a vertex or an edge.

***Programmatic demonstration.*** For a better understanding of the API of our temporal operators, we prepared a set of example programs[6] that show basic functionality and usage. Advanced examples, like the one in Figure 5, demonstrate the composition of multiple (temporal-) operators to answer specific analytical questions.

In an example scenario, we want to answer the following question: *In 2019, how did the minimum, maximum and average trip duration change per month for male and female users born after 1990 between stations located in different areas?*

We first use the *snapshot* operator (line 3) to retrieve all information about trips, users, and stations of the whole year of 2019. Since *snapshot* can produce dangling edges, we make sure to remove them by calling the *verify* operator (line 8). We further apply a *subgraph* operator with specific predicates (line 10) to filter for users born after 1990. At the end of our pipeline, we want to summarize our graph by calling the *grouping* operator with specific grouping key functions for vertices (lines 15-16) and edges (lines 22-24). Besides the predefined *label()* function, we also show the usage of a user-defined grouping key function (line 16). It calculates a map grid using latitude and longitude properties of our vertices. We further group the edges for each month of the year, separated by the two genders of the users. During this step, we also apply multiple aggregation functions to calculate the minimum, maximum, and average duration of trips (lines 28-30). Figure 4 shows the results of this GrALa program for two randomly picked stations.

The example illustrates the level of abstraction using our operators. A user does not have to care about the underlying graph data structure, operator implementation, or distributed execution details. Gradoop offers a variety of *DataSources* and *DataSinks* to read and write different kinds of data such as *.csv* files for data exchange or *.dot* files for graph visualization.

A visitor of this demonstration has the possibility to freely manipulate the provided GrALa programs. For example, it is possible to use *asOf* instead of *overlaps* at the *snapshot* operator (line 3) to specify a different time period. Further a user is able to use *pattern matching* instead of *subgraph* (line 10) to detect important graph patterns. Also the *grouping* operator can be parameterized by different pre- or user-defined functions to summarize different aspects of the data.

## ACKNOWLEDGEMENT

This work is partially funded by the German Federal Ministry of Education and Research under grant `BMBF 01IS18026B`.

## REFERENCES

[1] Charu Aggarwal and Karthik Subbian. 2014. Evolutionary network analysis: A survey. *ACM CSUR* 47, 1 (2014), 1–36.
[2] Orri Erling et al. 2015. The LDBC social network benchmark: Interactive workload. In *Proc. SIGMOD*.
[3] Tom Johnston. 2014. *Bitemporal data: theory and practice.* Newnes.
[4] Martin Junghanns et al. 2016. Analyzing Extended Property Graphs with Apache Flink. In *Proc. SIGMOD NDA Workshop*.
[5] Martin Junghanns, Max Kießling, Niklas Teichmann, Kevin Gómez, André Petermann, and Erhard Rahm. 2018. Declarative and distributed graph analytics with Gradoop. *PVLDB* 11, 12 (2018), 2006–2009.
[6] Udayan Khurana and Amol Deshpande. 2013. Efficient snapshot retrieval over historical graph data. In *Proc. ICDE*. IEEE, 997–1008.
[7] Christopher Rost, Andreas Thor, Philip Fritzsche, Kevin Gómez, and Erhard Rahm. 2019. Evolution Analysis of Large Graphs with Gradoop. In *Proc. ECML PKDD*. Springer, 402–408.
[8] Christopher Rost, Andreas Thor, and Erhard Rahm. 2019. Analyzing Temporal Graphs with Gradoop. *Datenbank-Spektrum* 19, 3 (2019), 199–208.
[9] Yishu Wang, Ye Yuan, Yuliang Ma, and Guoren Wang. 2019. Time-dependent graphs: Definitions, applications, and algorithms. *Data Science and Engineering* 4, 4 (2019), 352–366.

[6] https://git.io/JvSu0